Pre-Evaluating Application Integration Projects

Master's Thesis

Martin Haubrich

May 2005
Amsterdam, The Netherlands

vrije Universiteit amsterdam

Pre-Evaluating Application Integration Projects

Master's Thesis

Martin Haubrich
martin@cs.vu.nl
Student nr. 1142968

May 4th, 2005
Version 1.0

© Getronics PinkRoccade

vrije Universiteit amsterdam

Vrije Universiteit
Faculty of Sciences
Dept. of Computer Science
De Boelelaan 1081a
1081 HV Amsterdam
The Netherlands

Supervisors
Prof. Dr. J.C. van Vliet
Dr. P. Lago

Getronics PinkRoccade
Dept. Of Research and Development
Staalmeesterslaan 410
Postbus 57005
1040 CG Amsterdam
The Netherlands

Supervisors
Dhr. S.C. Chang
Ing. J.D. Gerbrandy, MBA

Preface

This thesis marks the end of my Master's at the Vrije Universiteit in Amsterdam. It has often been interesting and fun at the same time; with of course the usual pinch of stress. This thesis would probably not be here if it weren't for the people that both stood by me throughout the years and were of great help during the writing of the thesis itself.

For their help in writing this thesis, I would like to thank Hans van Vliet and Patricia Lago for their guidance from the Vrije Universiteit, Thiel Chang for embracing me in the PinkRoccade R&D family, Jelle Gerbrandy for his feedback (and hearing me out when I, once again, couldn't see the end of "all of this") and Mark Backer for the weekly meetings, where many useful comments surfaced. I had a great time at PinkRoccade, not in the least because of the fellow students there; thanks guys!

For their support and guidance throughout the years at the Vrije Universiteit, I would like to thank both my mother and Betty, who were always willing to hear me out, both when I had good and bad things that I needed out of my system. You even supported me when I had a maths exam, and that's impressive!

Last but certainly not least, I would like to dedicate this thesis to my father, whom I still miss every single day. Dad, whenever I felt I could no longer do it, I continued for you. I hope you are proud of me.

Martin Haubrich,
May 2005

Table of Contents

1. Introduction

The first chapter of any thesis outlines the basis for the subject. It explains what the subject is all about and why it is such an interesting topic to begin with. In this chapter, application integration will be explained, together with a short outline of why it is worth looking at from an architectural perspective.

1.1 Why Application Integration?

A typical enterprise today employs many different software applications, the amount of which can reach a few hundreds or even thousands in the bigger companies. These applications can do anything from keeping calendars to supporting salesmen in the field by means of a knowledge database. However, it is also very typical to see a demand for cooperation between these applications. You may want your calendar on your office desktop computer to cooperate with your PDA; you may want that a customer's change of address is not only updated in your sales system, but also in your customer relations system. Company mergers may suddenly require your personnel administration to merge with the other's. There are many scenarios in which applications need to cooperate. An application nowadays rarely operates in isolation.

Over the past decade or so, ERP systems such as Baan or SAP have become quite popular. The promise was great: solve all your enterprise's IT needs with one big application. But alas, many enterprises have experienced a greater IT need than could be solved by an ERP system. Niche markets with custom-built applications, but also many legacy applications have become the unwanted, but necessary islands of information. In fact, ERP systems are among the top of applications in dire need of cooperation with other applications. This is what application integration does: couple the data and/or processes of applications, which were independently built, in order to create additional value. This value can be, for instance, in enhanced data mining, faster throughput of your business processes, better support for your core business by back-office, etc.

So it is easy to imagine why one company would want to go integrate its own applications. Then there is also often the need to integrate applications owned by more than one company: business-to-business integration. This can become required as companies engage in a partnership, such as KLM and NorthWest, where it does not really matter at which of these companies you book your flight ticket. Their applications are linked in one way or another; they would not want human typists to enter reservations in both systems (this will lead to a lot of practical problems, aside from the double work-load). Companies may also want to have some form of automatic communication, especially in case one of them is a regular supplier for the other, say a parts supplier for General Motors. GM has many of those and it needs a lot of parts, so it expects its suppliers to accept automated orders, for instance.

Often, architects and developers are faced with the same challenges, whether they are dealing with business-to-business or enterprise (intra-business) application integration projects. Technically, it is often not too hard to implement some form of communication mechanism, but on an architectural level the problems can be quite challenging. One is faced with the meaning of the communicated messages, with the automatic invocation of functionality, etc. All of these have to be built and combined in a meaningful way; one cannot just add a communication pipe between components and transfer output from one of these as input to the next. With business-to-business application integration, however, many organizational issues will also come up. We can imagine some form of corporate confidential information that one business does not want to share with the other, or the lack of enthusiasm for taking on certain responsibilities that come with the integrated solution. The actual contracts that businesses sign amongst each other, however, are beyond the scope of this thesis. Instead, it will focus on architectural issues and solutions.

The cases that are put forth in chapter 5 will provide not just insight into the research, but will also serve as evidence that integration is indeed a very current topic for enterprises, software architects and developers.

1.2 How is it being done?

There is a wide array of different integration solutions available out there. Some aim just at making sure a series of applications work on a shared set of data, others link legacy applications with ERP systems to create added functionality. Some are done through simple batch file transfers at the end of each day, others require a live connection at all times to make sure there is no delay in the communication between the two applications. To create some order in this chaos, chapter 2 explains more about the high-level view we can take on integration projects. The literature as discussed there, aims at making a classification of different integration projects, by dividing them into certain *styles*. These styles will serve to clearly distinguish projects by means of their goals: what is being accomplished by coupling these applications?

Then there are the actual architectural *solutions* being employed to serve the need. As mentioned before, these can range from simple file transfers to complex service-oriented messaging systems. This all depends on the requirements put on the project and the style being aimed for. Chapter 3 elaborates on various often-deployed architectural solutions.

1.3 What makes it so hard?

Application integration is a form of software maintenance. When we look at the definition of perfective maintenance:

Perfective maintenance is an activity that we undertake to improve the maintainability, performance or other attributes of an application. Furthermore, perfective maintenance includes all changes, insertions, deletions, modifications, extensions and enhancements made to the application to meet evolving and/or expanding needs.

Integration is a result of evolving and/or expanding needs: we really need to get two (or more) applications to cooperate. The problem we often face, however, is the fact that an application that is to be integrated cannot be changed: a so-called legacy application. It is of prime importance to the enterprise, but there is no one available who really knows the inner workings of the product. Either the original contractor (as a company) has disappeared (bankruptcy, mergers, etc) or the people involved are retired. It may also be the case that we are facing a commercial application for which we do not have a contract to change the code. Either way, we are faced with a building block that cannot be changed. This is a problem, because it was never built to cooperate with another application, let alone that we can simply convince it to do so now.

On the other hand there is the fact that many applications that were separately developed do not have a mutual vision on the outside world. Applications work on models. These models form the basis for the data semantics within the application. It is rarely the case that two applications are based on the same model, let alone work with the same data semantics. One application may see a client to be a company, which happens to have a contact person; another may see a client to be a contact person, which happens to work for a certain company. We are faced with the problem that we have to transform the data to make sure that all applications are really communicating about the same. But not just the semantics are important here, also the syntax. For instance one application may set a person to be of the gender “F” where another really feels it should be “female”.

Once we have a series of applications communicating with each other (and we have made darn sure one knows what the other is communicating about) we still face the problems of lost, garbled, late and insecure communication, different technologies that were put to use, and monitoring the

(infrastructure-) solution, to mention a few. Some of these issues can (and should) be solved by deploying low-level infrastructure solutions; some should be addressed by higher-level architectural decisions. Either way, there are many things that influence an integration project.

This thesis aims at making it explicitly known which properties of an application make integration easier, harder or practically impossible. These properties can vary in their scope; we can look at architectural decisions taken in the past or run into more basic properties that helped the integration designer or programmer. Therefore, the main question is:

"What are the properties of an application that define its ease of integration?"

With these properties in mind, we can pre-evaluate the applications we want integrated, and *then* decide whether or not to commit ourselves to such a project. On the other hand, we can also imagine a situation where we can choose which of a set of applications to use (or acquire) for an integration project; knowing which properties to look at in such a case will help us make a proper decision.

Also, when we end up with a list of properties that ease the process of application integration, there might be consequences for architectures developed in the future. If I were to develop an entirely new application, one of the future requirements might be the integration of this application into a larger set. I could then take advantage of the list of enabling properties and incorporate exactly those I deem useful in that particular environment. A set of architectural recommendations for new applications with "integration" as future requirement is therefore a logical next step.

The remainder of this thesis will aim at solving the main question as put forth. But not before clearly defining from what point of view we will look at integration (chapter 2) and what the proven solutions are that can be deployed when solving the integration need (chapter 3). Chapter 4 will answer the research question, by putting forth a concerns and issues model, including a focus for each of the three styles as discussed in chapter 2. The cases that were used as a starting point for this thesis will be described in chapter 5, so as to illustrate the theory from chapter 4. Several concluding remarks will be made in chapter 6.

2. Classifying Integration Efforts

Before one can research the architectural level of integration, it is important to define what that level actually is. This chapter aims both at answering that question, as well as dividing integration projects in one of three classifications. Section 2.1 will discuss a three-layered model to form a basis for further reference when discussing levels of architecture. Section 2.2 discusses a classification put forth by Gartner Research, which helps to make a clear distinction between various styles of integration: there are different goals with different approaches involved.

2.1 The Architectural View

Software development can be seen from many angles; there are often organizational issues involved (e.g. project management), but one can also focus on the algorithms involved (e.g. validation of algorithms). Then there is also the issue of components and their connections: architecture. In this section, a model will be discussed which enables us to divide an architecture in different layers, with different views. Some issues are more low-level than others, and it is important to get a feel for this layering of issues. Along the way it shall be pointed out at what layer this thesis is aimed.

Wilhelm Hasselbring [Has00] discusses the fact that enterprises are often "vertically fragmented" into separate organizational units or departments. Each of these used to be rather independent of the others when it came to the development and deployment of information systems. For instance, an enterprise had separate departments for maintaining track of inventory and for sales. Human interaction made sure that they did cooperate, but their information systems were separate entities within the enterprise. This has lead to a great diversity of applications within a single enterprise and only with the emergence of ERP systems did companies start to create an enterprise-wide IT vision. ERP systems, however, have not proven to be the solution, and many "islands of information" still exist within enterprises; the so-called legacy systems in use have proven too hard to simply replace. Application Integration in this case would be aimed at coupling the systems of both inventory and sales departments to make the cooperation and communication between the respective departments run smoother, faster and more accurate. The need for the actual human communication should dissolve, and whenever a sale is made, it can be both immediately verified with the current inventory and either subtracted from stock or ordered anew.

Hasselbring continues by discussing that within said organizational units, or departments, the information systems can be "horizontally fragmented". Each unit may be structured within three architectural layers, as described in the following:
The *business architecture* layer defines the organizational structure and the workflows for business rules and processes. It is a conceptual level expressed in terms meaningful to actual users of application systems. At this level, we are faced with issues such as business objects, business processes and business services. Most of these could even be defined outside the scope of an application: perhaps in a mission statement or in workflow diagrams that employees have to adhere to. When two companies decide to cooperate, it is often the case that these companies will have to search for ways to connect their mutual business models and processes. The integration of these will most likely manifest in lower layers, but an actual implementation at this level will not surface; applications will have to be modelled after the new business model. Reaching an agreement on business-to-business cooperation and the creation of resulting workflow diagrams is beyond the scope of this thesis. It will be assumed that integration projects will either start without the need for this (as is the case in Enterprise Application Integration, as opposed to business-to-business integration) or after reaching agreement on this level. Instead, the focus will be on the properties of the next layer:
The *application architecture* layer defines the actual implementation of the business concepts in terms of enterprise applications. At this layer, it is the central goal to provide the "glue" between the application domain described in the business architecture and the technical solutions described in the technology architecture. At this level, we are concerned with the original, or "source",

applications as components within the integrated, or "target", solution. As can be seen from the research question itself, this is exactly the level that will mostly be focused on. We will be looking at properties of the source components that make it harder or easier to create a target solution. We cannot fully ignore the business architecture layer, but we can try and abstract from it in this case. Then there is the *technology architecture* layer; it defines the information and communication infrastructure. When one integrates at this level, it is done by deploying technical solutions such as Message-Oriented-Middleware (MOM), shared databases, or batch file transfers.

In reality, these horizontal layers of organizational units cannot always be seen in isolation from one another. Especially when considering the business architecture, we cannot see the sales department separate from the inventory department. The human interaction between these two, as mentioned before, is a part of exactly this layer. It's just that the lower layers do not support the link, and thus need to be integrated. This is exactly where application integration pops up. Figure 1: Architectural Layers and Integration illustrates all of these concepts.

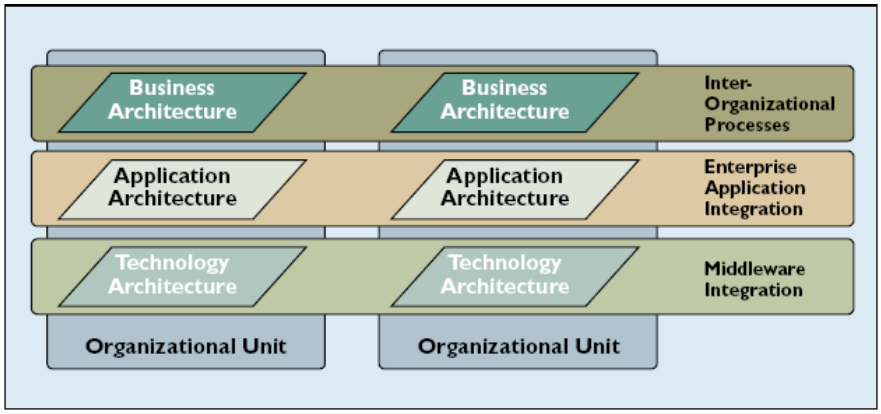

Figure 1: Architectural Layers and Integration

The horizontal bars that spread across organizational units are the layers where actual integration can take place:
Inter-Organizational Processes are usually business-to-business integration projects, such as the coupling of a supplier and a client in an automated fashion. Sometimes lower-level integration may have an impact on this layer. For instance, when the inventory and sales departments are coupled at the application layer, the human interaction is no longer necessary and the workflow changes: the step where an employee of the inventory department enters the mutation in stock after a sale may now be achieved in an automated way.
Enterprise Application Integration is aimed at coupling separate applications in order to make them cooperate. This is done at the application architecture layer. In sections 3.2.2 and 3.2.3, Service-Oriented and Event-Driven Architectures will be discussed respectively. Since these are ways of looking at an application, instead of actual technical solutions, they fit into this layer. Usually this coupling of applications is achieved by some kind of messaging service or other technical solution at the technological architecture layer. As said before, coupling applications at this level might have consequences for the topmost layer; it might affect the business processes, even within one enterprise.
Middleware Integration addresses the syntactical level of integration: it employs techniques to realize the infrastructure such as messaging or database gateways. As mentioned before, this is the enabling layer for actual Enterprise Application Integration, which itself addresses also a semantical level. There is already a lot of literature on this, including entire books with patterns (which are collections of experience) for messaging systems [Hoh03]. This thesis will not truly expand on the technologies that can be used, but it cannot completely forsake the issue either; section 3.1 touches on some of the basic architectures that can be deployed at this layer.

The thesis will mostly be aimed at the application architecture layer: we are looking for properties of applications that make them easier/harder to integrate. However, some application integration projects will have effects on the business architecture layer, so we cannot ignore issues such as business process management either. Also, the lowest layer, technology architecture, will often be

touched upon as enabling technologies are discussed or proposed, depending on the properties of the source applications.

2.2 Gartner's Styles

Application Integration has previously been defined as "coupling the data and/or processes of applications which were independently built, in order to create additional value." This includes a lot of different projects. The aim in this section is to make a clear classification of projects, so that we know what we are dealing with in each case. Some projects aim at a consistent, shared dataset between applications, others are built by actually using the functionality of other applications. In each of these cases, different properties of the source applications will play roles of different magnitude. For instance, if two applications are "only" to work on the same dataset, their mutual functionality need not necessarily be exposed to one another, let alone the outside world.

Gartner [Alt01] uses a classification of integration projects, based on three different categories: Data Consistency, Multi-Step Process and Composite Application. They differ from one another at key points, and each of these will be discussed in the subsequent sections. All of these styles fit into the application architecture layer as discussed in section 2.1.1.

2.2.1 Data Consistency

Data consistency problems arise from the level of data redundancy that characterizes all enterprises. To address this redundancy, enterprises have historically written batch programs that 1) extract data from the data stores (files or DBMSs) used by the source applications, 2) transform the extracted data to match the data model used by the target applications, 3) if necessary initiate the transfer of that data to the computers that host the target applications and 4) load that transformed data into the data structure used by the target applications. This is an obvious tedious and error-prone process; we are dealing with customized solutions, where the designer or programmer has to have a lot of knowledge about both source and target applications. The solutions are rarely scaleable, and latency is definitely an issue.

This "old style" approach to data consistency problems, however, will continue to be the preferred choice in some situations, where the aforementioned issues are not considered issues at all. For instance, when two applications already operate on a very similar data model, and where latency is not an issue, it may be acceptable for the applications to synchronize their data once every 24 hours. There are many other possible solutions to the data consistency problem, including shared databases or full-fledged integration broker suites from vendors such as BEA or Tibco.

Either way, the general, unifying goal of all data consistency projects is for redundant data in multiple systems to agree [Fri04]. There are various ways of accomplishing this: immediately or delayed transfer of either mutations or entire data sets. It can even be accomplished without the various applications involved "knowing" that they have been integrated. Some form of monitor could be installed to dispatch changes in the data set to other applications once they are detected. The applications involved do not rely on one another; they are often both physically and logically independent, and the messages (files, change logs) are sent asynchronously; the sender does often not expect an answer.

Consider a company selling office supplies. It has, amongst others, a department for sales and one for customer relations. The sales department sends its representatives to various offices throughout their operational region in order to open negotiations and make actual sales. The customer relations department is, amongst others, tasked with spreading commercial folders. Every time a sale is made, that particular customer is added to the database of customer relations, so as to send the folders their way as well. Salesmen sometimes offer customized "regular customer discount" to pull customers over the line, into the sale. They do so by checking the sales database and seeing if they are actually dealing with a regular customer or not.

The board of directors has decided that sales have to improve, and one way is to standardize this discount for regular customers. However, it would be a waste of time to wait until a salesman is on-site before offering this discount. They decide that customer relations should start sending coupons with discount offers to the regular customers. This means that both departments will now have to start sharing the data they have on customers and sales; cooperation is key in keeping this discount scheme in check. We wouldn't want one department to "under- or overbid" the other, and we sure don't want customer relations to send these coupons to "customers" who never bought anything before. One way or another, the data that both departments operate on will have to be synchronized, or even shared. The technical solution aside, the original and future situations can be modelled as is shown in figure 2: Data Consistency Example. This is a typical example of integration by means of the data consistency style.

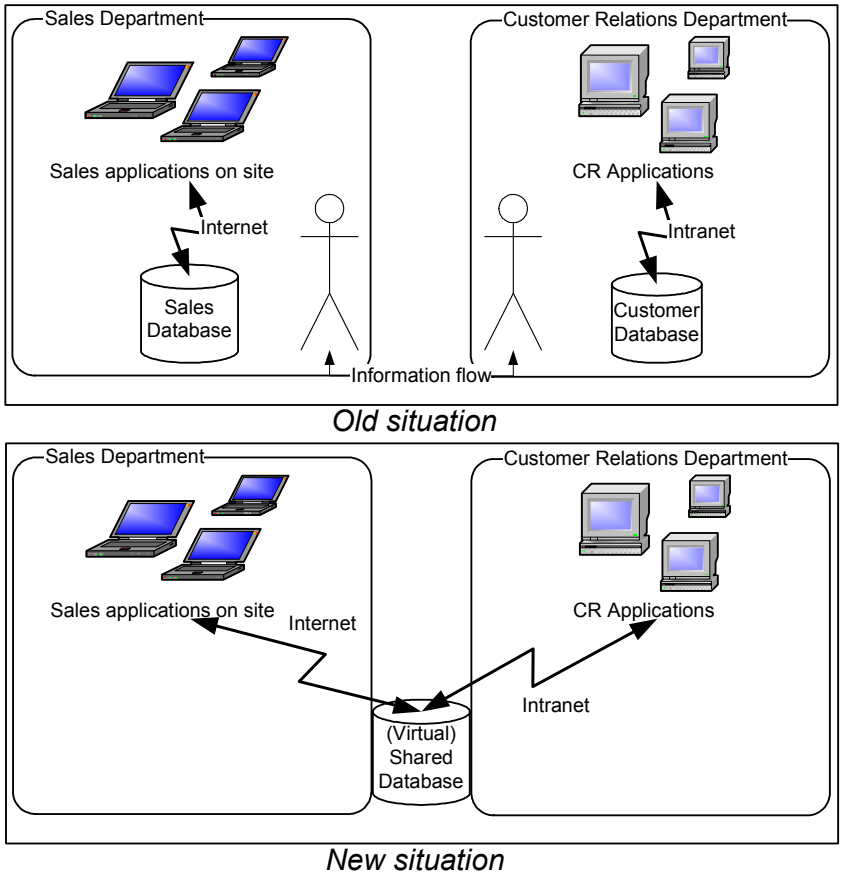

Figure 2: Data Consistency Example

This style is the most common pattern; the left hand must know what the right is doing. Many companies have seen the need to make sure that their various organizational units are working with the same knowledge of and the same view on the outside world (customers, inventory, suppliers, etc.).

2.2.2 Multi-Step Process

The premise of the multi-step process integration style is that a transaction (such as a payment, trade, purchase order or change to a customer's telephone service) is entered only once. Thereafter, it proceeds in an automated fashion through its lifecycle, potentially involving dozens of steps in various locations, without re-keying the data. This reduces input errors, improves service delivery, and usually shortens the time to completion. The lifecycle may be managed by a business process manager, depending on the variety in each transaction's route taken through the applications that are involved in the whole of its processing. If there is great variety, depending on various rules, it

makes sense to make such a manager responsible for the routing of each transaction. If there is a standardized route to be taken, or very little variety, it may be useful to have each transaction manage its own route or to have the applications decide where to send the transaction next.

Transactions may be sent immediately or in batch-style, and the dispatching is done in an asynchronous style, much the same as in the data consistency style. This time, however, one transaction maps onto one business process; its steps map on the steps in the business process. Another difference is the logical dependence of the applications involved in processing the transaction: all of them need to have semantic knowledge about the transaction itself and often also know about at least the next step being taken in processing it, and often also the previous one. If one step in the process fails, it is most likely that the entire transaction fails. This implies the logical dependency. Physically, however, the systems are not dependent on one another: one application processes the step it is responsible for, and subsequently yields the transaction to the next step, which may be another application, the process manager or even possibly the same application once more, but it need not know that per se.

A typical example of a multi-step process is when a series of tasks is triggered by some (business) event. For instance, if one were to go to the doctor with certain symptoms of a minor illness, the doctor first has to make an assessment of the situation and then decides whether or not to prescribe a certain medicine. With the prescription in-hand one is then supposed to actually pick up the medicine at the local drug store. The doctor will make a note of the prescription on the patient's record, and from then on it is assumed that the patient actually takes the medicine. We all know how legible doctor's handwritings are, so a more automated process between doctor's practice and drug store instead of the hand-written prescription might be a good idea. The trigger is the prescription of the medicine. The doctor (or the assistant) enters the prescription into the system, which communicates it to the drug store. When the patient arrives there, the medicine is most likely already packaged, labelled and ready to go. Once it is picked up, the bill goes to the insurance company, and the patient will live happily ever after. Obviously several organizations are involved, but we are dealing with but one client all of the time – the patient. There are also several steps that can be identified in the entire process; figure 3 depicts a simplified model of this situation. This is a small, but recognizable example of a multi-step process. Obviously, this example spans several organizations, but that is not always the case; multi-step processes within an enterprise are also possible. Think about the events that will occur just within the insurance company: your policy will be updated in many ways (e.g. no-claim and own risk) and, of course, a permanent record will be made of this bill. It is very unlikely that someone will actually sit at a keyboard and type your name over and over again in different applications and/or records.

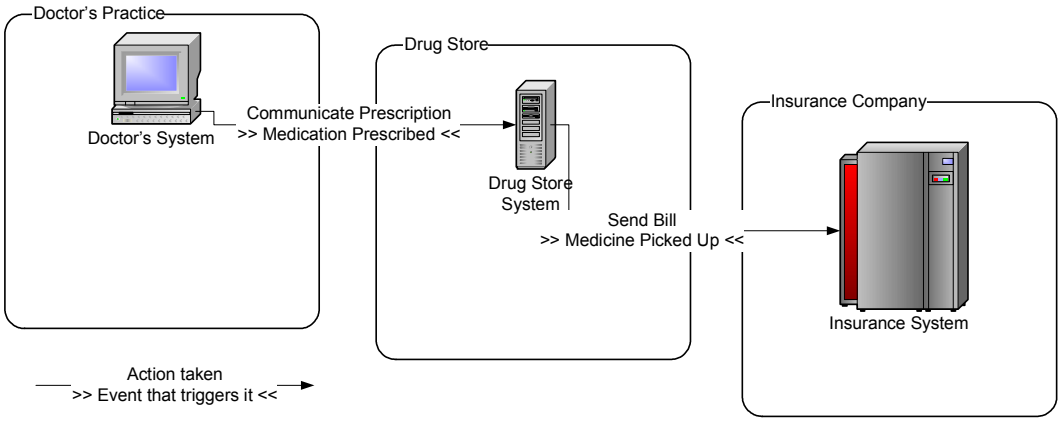

Figure 3: Multi-Step Process Example

2.2.3 Composite Application

Composite Applications involve immediate interactions among two or more semiautonomous applications, working in concert to execute a single step in a business process. This style ensures that legacy systems can still be used, by means of exposing its functionality through a request/reply interface, for instance.

In contrast to the previous two styles, the composite application style is built up through synchronous (i.e. request/reply) interactions between a requesting program and the programs responding to those requests. The recently emerging hype around "Web Services" fits into this category: it enables companies (or even departments) to work together by sharing their respective functionality (services) in a request/reply fashion. Other differences include both physical and logical dependence, because of the direct, synchronous calls being made and both requests and results requiring correct interpretation.

Composite Applications are usually built up of one application that combines the functionality of two or more "lesser" applications, adds some logic of its own, and thus offers more comprehensive functionality than any one of the original applications does. A good example is an on-board navigation system for a car. The system knows its current location (by being subscribed to the GPS system for example), and it is used by entering a desired destination. But not only does it communicate with a nearby roadmap station, it also queries nearby traffic stations for updates on heavy traffic. Both of these base stations (roadmap information and traffic information) can be run by the company itself, by partners, by third parties or any combination thereof. The on-board system subsequently combines the information it has acquired and then provides the driver with appropriate directions. These may be based on user preferences, of course, but that is beside the point. The idea is that the on-board system is an application that calls upon the services of two other applications: those for roadmap and traffic information. In essence, it forms the integration block between the two. This situation is modelled in figure 4.

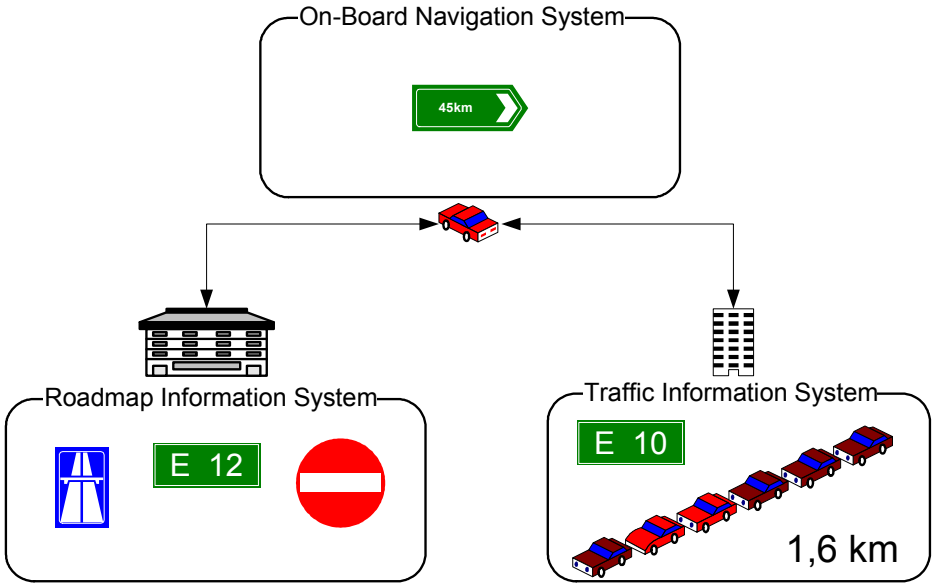

Figure 4: Composite Application Example

2.2.4 The Styles Combined

The previous sections each explored one of three integration styles. Figure 5: The Three Common Integration Patterns combines all of them in one picture and once again names those issues that separate them from one another.

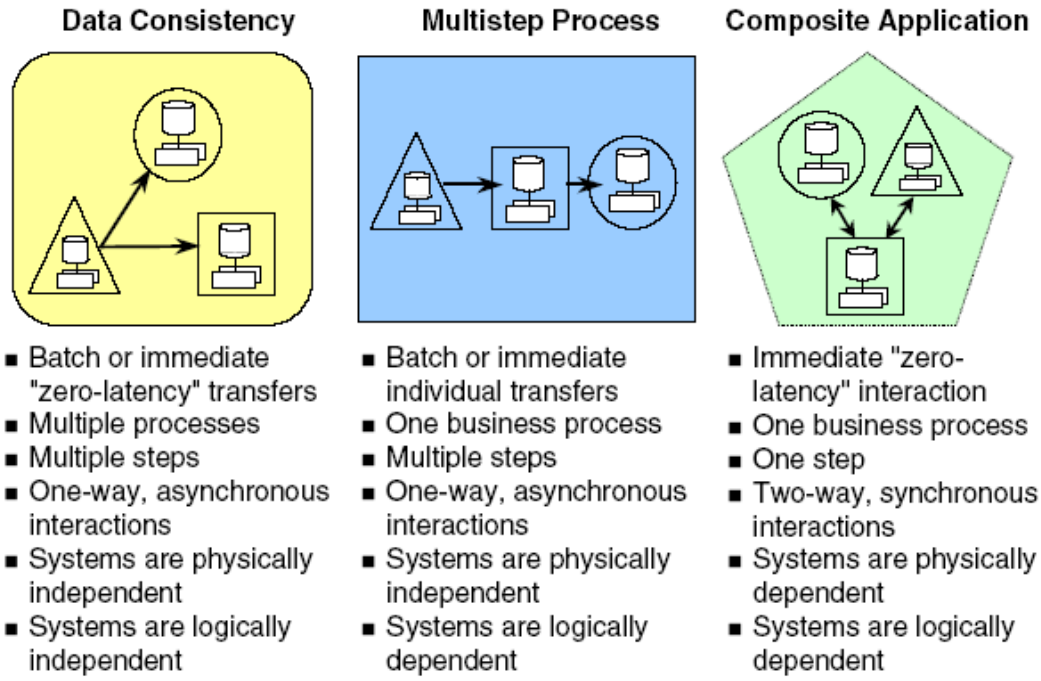

Figure 5: The Three Common Integration Patterns

Not every integration project, however, can be placed within one of these three niches. There are many cases where a combination of styles will be required to reach the desired functionality or added value. This is no problem: the styles are easily combined. The styles do help in this case to separate the different couplings being developed and to identify the issues the developers need to keep an eye on.

Consider a more elaborate example. The navigation system from the previous section on the composite application style is being expanded in two different ways. First, it turns out that one company now has a monopoly on the road map services in Europe. It decides to implement the data consistency style to be able to update its road maps and share the updates throughout Europe. Second, the navigation system's producer has thought of a profitable new addition: emergency services dispatching. It has developed a way to detect a collision much like airbags do. Once a collision is detected, the location of the vehicle is dispatched to a nearby emergency post. From there, a phone call is attempted to the registered (mobile) number of the owner of the navigation system. If the call is answered with the need for emergency services such as police or ambulance, or if the call fails entirely, the emergency post will dispatch the necessary services. This is a multi-step process, this time with a human step involved: the phone call. The entire new situation is modelled in figure 6, in which all three styles are now combined to come to a full-fledged integrated solution. Not all of them will be built by one company in this case, but that is not necessarily true for all mixed style solutions.

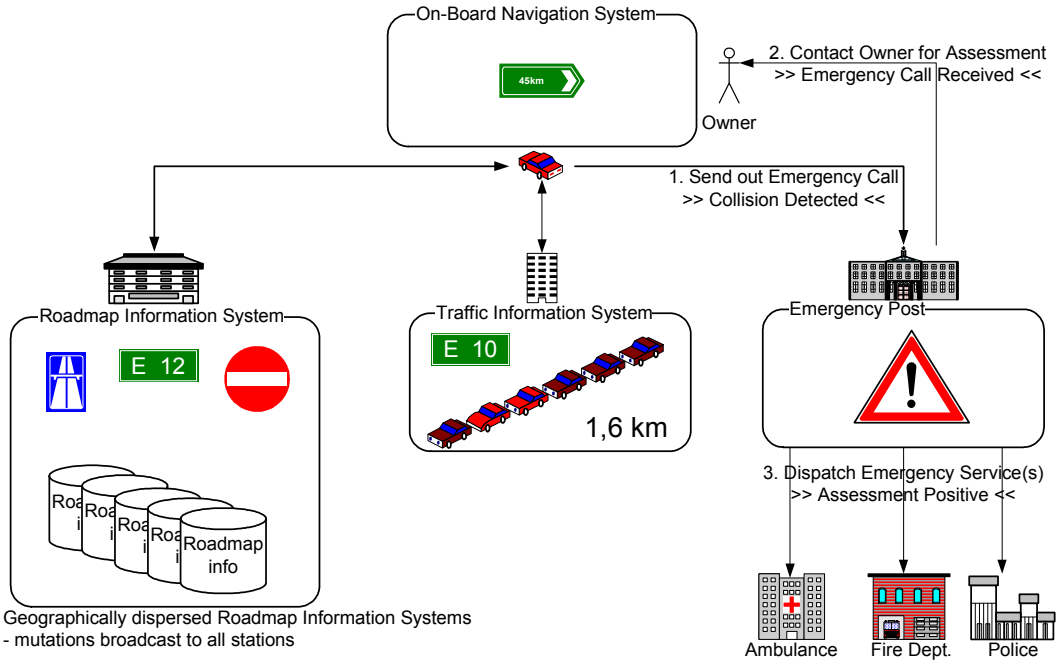

Figure 6: Mixed Style Example

So far, the models have been rather graphical, not technical in nature. This is because the focus was not on the implementation, but on the style in use. The mention of a broadcast in figure 6 is actually a bit too accurate, because the updates could just as well be batch-sent once a day or gathered in one central location before the new data-set would be transmitted once a week. Either way, the aim was to divide different approaches to integration into different styles. The style used depends on the goal of the integration project and will impact the solutions that will be deployed. It has also been pointed out that not all integration projects fall into just one style category; it is often the case that a mix of styles can be identified in the project. With this in mind, the architect or developer can focus on different issues with each style instance.

3. Integration Architectures

The three styles of integration as discussed in section 2.2 were all abstract interpretations of the integration problem at hand. Indeed, the actual implementation of the application's couplings was mostly abstracted from. But any problem requires a solution, and in the case of integration there are many that can be deployed. Not all of them are equally suitable to all tasks, but all do have their vices and virtues. This section will discuss several low-level technological solutions in section 3.1: Middle-ware as Simple Architectures and subsequently discuss two higher-level architectures in sections 3.2 and 3.3 on Service-Oriented and Event-Driven Architectures respectively. Section 3.1 would fall into the technology architecture layer, whereas 3.2 and 3.3 would fall into the application architecture layer, since they describe a more semantical way of looking at integration solutions.

3.1 Transport Architectures or Middle-ware

Because they are old and simple, people may tend to ignore these low-level solutions, but though old and simple, the four solutions as discussed in this section are still elegant, feasible and useful. They also form the basis for other, more elaborate schemes of integration architectures. Therefore, file transfers, shared databases, Remote Procedure Invocations and message-oriented middleware cannot be ignored when discussing the integration of software.

3.1.1 File Transfer

Hohpe and Woolf define the File Transfer Pattern in their book Enterprise Integration Patterns [Hoh03] as follows:

"Have each application produce files of shared data for others to consume, and consume files that others have produced."

This technical, simple architecture is aimed at exchanging information. One application does so by exporting the necessary information into a standard format. This format may be customized for the integration solution at hand, or it may be a more common standard. Either way, the result is a file with all necessary information, which can subsequently be transmitted to other applications, which should then import and filter the information from the file itself before processing it. The nature of this scheme makes it useful to have the files produced and shared at regular intervals, such as during nightly batch sessions. Figure 7 illustrates this scheme.

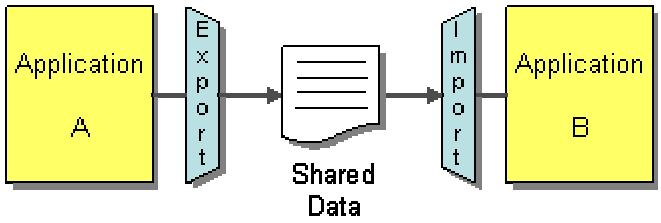

Figure 7: The File Transfer Scheme

The file will commonly include mutation data, such as new additions or changes to the database, which application A has gathered during the day and which application B should process as well. It may also be an order list from the General Motors company to one of its parts suppliers; the file will the simply contain the order(s).

3.1.2 Shared Database

The previous scheme introduced a form of latency that is often unacceptable. Sometimes, applications have to work on the same dataset throughout their operational periods. The Shared Database offers the solution. This Pattern is in [Hoh03] defined as:

"Have the applications store the data they wish to share in a common database."

The "common database" need not necessarily be one physical database; it may also be a virtual shared database in the sense that a communication mechanism ensures that all physically distinct instances of the database are kept in sync. Transaction management play a big role in this case, but transactions are fairly common for any DBMS. The scheme is illustrated in figure 8

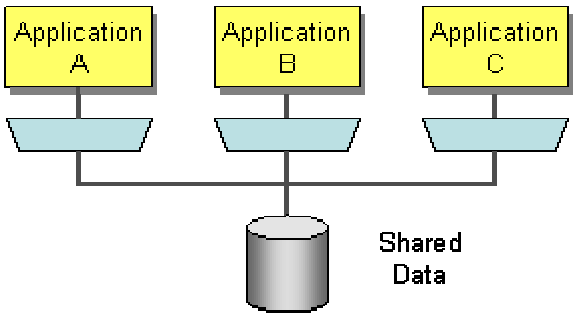

Figure 8: The Shared Database Scheme

In this illustration, it is implied that each application's database is replaced by a router component, which reroutes the queries to the actual shared database. It simulates a local presence, but ensures that the dataset is common to all applications involved in the scheme. As mentioned before, the router components can also take another role: that of keeping all separate databases in sync, in that case the local databases may still be physically present. A common occurrence of this scheme is when two departments, such as the sales and customer relationship departments of section 2.2.1, are going to have to share whatever information they have (see figure 2).

3.1.3 Remote Procedure Invocation

However, unlike the previous two schemes, which only share information, it is also often the case that functionality has to be shared. C/C++ RPC and Java RMI are both examples of Remote Procedure Invocation. [Hoh03] defines the Remote Procedure Invocation Pattern as follows:

"Have each application expose some of its procedures so that they can be invoked remotely, and have applications invoke those to initiate behaviour and exchange data."

It applies the principle of encapsulation to integrating applications. If an application needs some information or functionality that is owned by another application, it asks that application directly. If one application needs to modify the data of another, then it does so by making a call to the other application. Each application can maintain the integrity of the data it owns. Furthermore, each application can alter its internal data without having every other application be affected. The principle is outlined in figure 9.

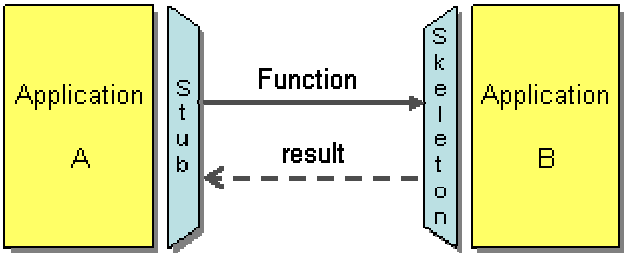

Figure 9: The Remote Procedure Invocation Scheme

The scheme was once designed to emulate local procedure/function calls, as is common in procedural and object-oriented programming languages such as C, C++ and Java. It will work fairly well in cases where the original application was already built to accommodate the scheme, or where it is easy to build an additional layer to accommodate the remote calls and transform them into local ones. It will, however, not always work because of the language-specific issues involved. If application A were written in Java and application B in C, the communication would not be as easy and straightforward as depicted in figure 9. However, the scheme is applicable in many situations, and the service-oriented architecture as discussed in section 3.2 is a generalization of it. This scheme falls into that category.

3.1.4 Messaging

The most flexible technical approach to integration is Messaging. It accommodates both information sharing and the sharing of functions and services; it simply abstracts from the goal at hand and focuses on the communication mechanism instead. Again a definition as taken from [Hoh03] on the Messaging Pattern:

"Have each application connect to a common messaging system, and exchange data and invoke behaviour using messages."

These messages can obviously both exchange data (share information) and invoke behaviour (functions, services). The content of the message is what defines its purpose. The lion's part of the book Enterprise Integration Patterns describes patterns for use with messages. It elaborates on Message Brokers, Messaging Gateways, composing messages and interpreting them. The basic idea is illustrated in figure 10.

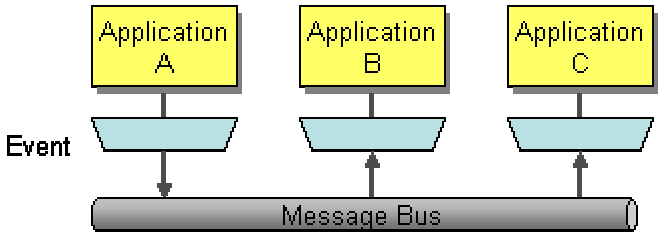

Figure 10: The Messaging Scheme

The sharing of information is quite easy with a messaging scheme in place; the changes or updates are transmitted "live" to the other applications. In this case, the Event as modelled in figure 10 would be the notification of the update. Basically it can be implemented by means of a "fire-and-forget" scheme, where the update is broadcast to the message bus (or by means of a publish-subscribe mechanism) after which each application is free to do what it wants with the notification.

The sharing of functionality here is a bit different from the previous scheme, Remote Procedure Invocation. In that case, the call is made and the caller waits for a response. The thing is that this call could potentially be "change the address of contact person "Betty" to <address>". The response could be an acknowledgement or a failure message. Either way, the caller is not going to do much with the response; it expects one, but should not wait for one. Remote Procedure

Invocation is synchronous, whereas the entire Messaging scheme is asynchronous. The message is sent about the update, but the caller continues processing any tasks at hand, without wasting precious time waiting for an answer. And when it does have work to do when the answer comes, it can simply continue where it left off. Either way, it's not idly standing by. For instance, imagine an application that asks around for roadmap information. It could send out five queries and pick the one with the fastest response. With Remote Procedure Invocation, it would have to try a call five times, wait for five answers (in the same sequence before calling the next!) and compare the results. A messaging system would have had the answer already: it sends out five messages and waits for the quickest reply.

In both cases, both calls (i.e. requests) and replies can get lost somewhere along the way. In the case of Remote Procedure Invocation, the calling application itself is responsible for handling time-outs and retries. In the messaging scheme, it is often a separate component that ensures exactly-once or at-least-once delivery of the request and reply messages respectively. In the latter case, applications can abstract from the infrastructure and focus on the task at hand.

Messaging is the scheme with the most potential; it is flexible, scaleable and a lot of patterns can be customized and applied when needed. These patterns can solve the problems arising from lost messages, error messages, slow servers etc. Nonetheless, the previously discussed schemes of File Transfer, Shared Database and Remote Procedure Invocation all have their virtues. In some cases it is just easier to deploy a simple solution instead of a full-fledged messaging system.

3.1.5 The Transport Architectures Compared

The four previously mentioned transport architectures form the backbone of communication between applications. They are the basis on which full-fledged solutions can be based, or they are used as such, without further ado. This section sums up the major differences between the four, as shown in table 1.

File Transfer	**Shared Database**	**Remote Procedure Invocation**	**Messaging**
Periodical batch-transfer	Constantly synchronized information base	Functionality defined in "contract", implementation remote and custom	Use defined during development: project size increases greatly
Large data capacity	Various technical implementations possible	Standard type data communication	Highly flexible
High latency	Low latency ("live")	Synchronous	Asynchronous
Scaling often not considered	Bad scaling	Average scaling	Good scaling
Typically used to share information on a regular interval	Typical basis for common, shared information	Typical for sharing functionality	Open-ended: goal depends on implementation
Fits in with the Data Consistency or possibly Multi-Step Process Style	Fits in with the Data Consistency Style	Fits in with the Composite Application or possibly Multi-Step Process Style	Fits in with all Styles

Table 1: The Transport Architectures Compared

3.2 Integration Architectures

Section 3.1 focused on the communication and transport architectures, which are in fact but technical solutions we can place in the technological architecture layer of figure 1. In this section, a few higher-level views on integration solutions will be discussed. These can be placed in the application architecture layer and can be seen as a view on the implementation itself. The typical technical implementation of all of them is based on messaging, or in some cases on file transfers, but in that case the file can be seen as a large message. The architectures in this section are based on open connections, with the applications almost always ready to receive communications and act upon them. Messaging, therefore, makes for the most logical implementation model. Not in the least because of its flexibility.

3.2.1 The Broker Architecture

The broker is a somewhat aged ideal vision on the solution to an enterprise's integration needs. The idea is that there is a central hub, where all communication can be directed to, which decides what to dispatch to which application and where thus all functionality and information can be shared. This central hub would provide a central point of maintenance and dispose of all ad hoc connections in the company. All applications would be communicating about the same enterprise-wide data model and information was shared completely and consistently.

And there would be one point of failure, one bottleneck. Such a hub-and-spoke technology might work for smaller scaled integration problems, but the idea of the broker is to serve a great variety and number of applications. Figure 11 illustrates the idea.

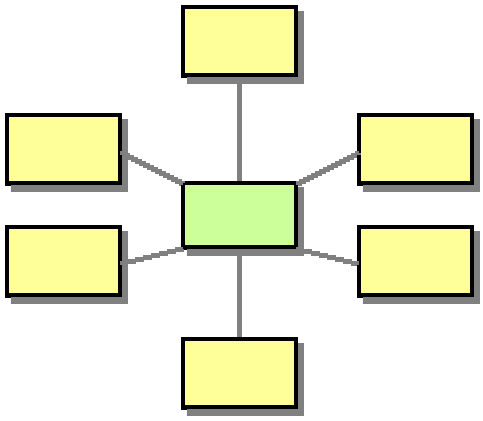

Figure 11: The Integration Broker

If the Integration Broker is considered at all, nowadays, it is often a message broker which transforms and routes messages to their intended destination. And, in order to scale, a federation of these brokers is installed. The brokers are then connected to one another, with each broker serving a local section of the whole network. Linthicum [Lin99] calls this the Multihub Configuration. Another option he mentions is the Bus Configuration, with a message bus where applications directly connect to. A Message Broker is also directly attached to the bus, providing broker services when needed, but not serving all messages. This scheme is closely tied to figure 10, where no broker is depicted. If one of the three applications were a broker in that particular figure, "listening" to the bus for appropriate messages to be routed or stored, for instance, figure 10 would depict Linthicum's Bus Configuration. This is useful in the case where the broker plays a smaller role. Both variations remove the single point of failure, sometimes referred to as the "Über Broker" [Hoh03].

3.2.2 The Service-Oriented Architecture

Service-Oriented Architecture, or SOA, is a client/server design approach in which an application consists of software services and software service consumers (also known as clients or service requesters) [Nat03]. SOA differs from the more general client/server model in its emphasis on loose coupling between software components and in its use of separately standing interfaces. SOA principles are put to use during design, development and deployment. The fundamental intent of SOA is the non-intrusive reuse of software components (services) in new runtime contexts.

The service interface is the design essence of SOA. It defines the contract to which the service will comply, as long as the service requester does the same. For instance, as long as the requester provides the correct Social Security Number, employer data and security credentials, the service will reply by providing a summary of that person's year earnings and medical expenses. The implementation of the service itself is often a "black box": the requester does not (and need not) know anything about the internal workings. This implies that we can wrap a service interface around a legacy system and use it as part of a Service-Oriented Architecture. The legacy system is "just a black-box implementation" of the services as described in the interface.

The trick is to not only find the correct granularity of the service components, but also to design them in such a way that they can actually be reused and/or replaced. If an application is built up by means of the SOA principle, but where all of its components are so very logically locked into one another, it may render the entire application monolithic, in spite of the use of SOA principles. This thesis, however, is not on proper SOA design and development; there are many great books out there that will serve as a guide to aspiring Service Architects. This thesis is on integration, and integration combined with SOA principles leads to situations such as depicted in figure 12.

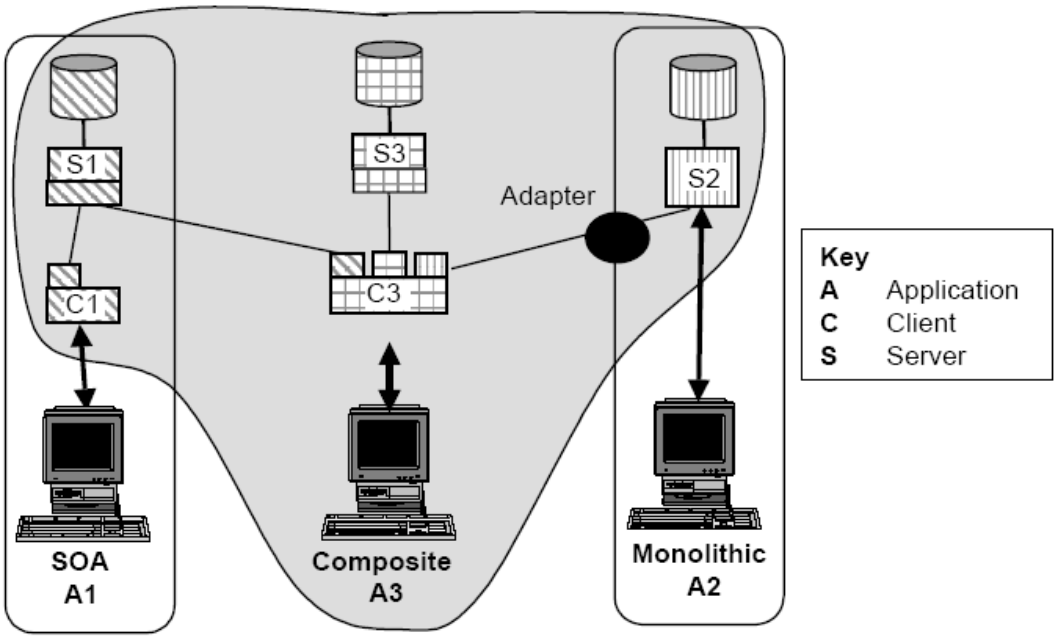

Figure 12: Composite Application Based on SOA

The figure depicts a composite application style (section 2.2.3), where A3 is the application that combines both A1 and A2 functionality. A1 is an application built on SOA principles, with S1 being the service provider component. A3 simply requests those services and makes use of the replies. A2 is a legacy, monolithic application, not built on SOA principles at all. However, as mentioned before, it is possible to create some form of wrapper that translates S2's functionality into services and publishes those in an interface. A3 makes use of that adapter's services and through it, of S2.

In short, Service-Oriented Architecture is based on the dual principle of modularity – breaking big jobs into smaller tasks – and encapsulation – using a clearly defined interface to encapsulate the module internals from outside contact [Nat03]. As long as the interface "contracts" are respected, each component can be developed, tested and modified with a minimum of dependencies and coordination overhead.

Application integration, on the other hand, means making independently designed applications work together [Sch03]. "Independently designed" implies that development teams select their respective technologies (such as programming languages, development tools, database management systems, operating systems and middleware) and information models (such as semantics and data, object and process models) without referring to the design decisions of other teams. Developers were autonomous upon building the software, but subsequent teams must work on making the completed applications work together.

All of this implies that "Service-Oriented development of applications requires designing for application integration while the applications are being built" [Plu02]. This will especially enable the Composite Application style (section 2.2.3) because of the modularity and the encapsulation of functionality by means of a request-reply mechanism. Of course, services that update data (in light of the Data Consistency style) are also easily built, but may be a bit overkill (the service reply is a bit over the top). A service-oriented Multi-Step Process is also feasible, but in order to truly enable the stepwise processing of an event, the replies should be gathered by a process manager, which can keep track of the progress of each event and steers the path it takes through the process chain.

3.2.3 The Event-Driven Architecture

Event-Driven applications sense and respond to business events, which are relevant changes in the state of the world [Sch03a]. The word "relevant" is of importance here; the developers of event-driven applications define and model those events that are deemed relevant to the business and the application in particular. An event may be an incoming order from a customer, or the hiring of additional personnel, for instance. Either way, the event triggers a certain action or series of actions that have to be performed. So unlike with Services, the application does not make a request for a service and waits for a reply, but it stands by to receive events, to which it responds by firing off a series of processing steps. These are also known as "pull" and "push" mechanisms respectively. Most of the time, an event-driven application will subscribe to other applications, which themselves publish events. The subscriber deems these events as relevant and wants to be notified when such an event occurs so it can take appropriate action.

In comparison, SOA is often seen as a linear path of execution where the service requestor steers the flow of events. Event-Driven Architecture (or EDA), on the other hand, supports dynamic, parallel, asynchronous flows through a network of modules, and that flow is even determined by the recipient based on the event itself. Figure 13 depicts these differences.

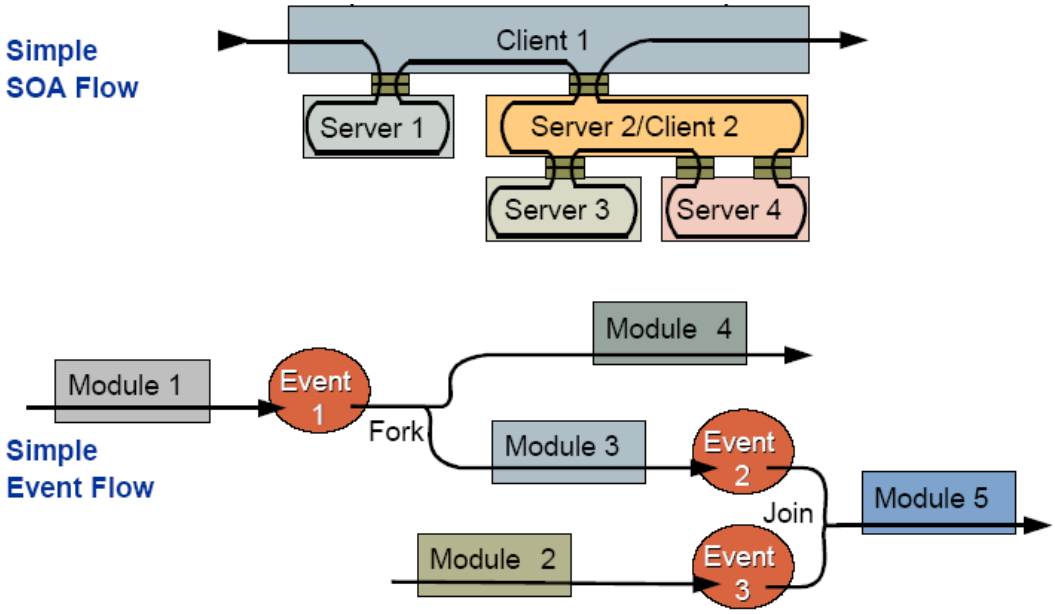

Figure 13: Simple SOA and EDA Module Relationships

As was said in the previous section on SOA, requests have a reply back to the client. But that is actually not a requirement. If there is no response, it is still a SOA as long as the connection is 1-to-1 and the client specifies the action to be taken. In contrast, an EDA can publish its events to many subscribed clients, and it just sends a notification, it does not say anything about the appropriate response, it leaves that to the subscriber.

The Composite Application style (section 2.2.3) is poorly supported by EDA; a composite application makes use of the functionality of other components and thus implies a request/reply mechanism. This, of course, can be realized by means of events, but it makes no sense when we have SOA standing by. Both Data Consistency (section 2.2.1) and Multi-Step Process (section 2.2.2) styles are well supported, due to incoming events without the need for actual replies. Data consistency can be achieved relatively easy by gathering mutation-events and applying them to the actual data set with a certain interval or even immediately.

3.2.4 The Integration Architectures Compared

The architectures that were mentioned throughout section 3.2 can all be placed in the application architecture layer of figure 1. They can also all be implemented by means of the transport architectures of section 3.1. Table 2 sums up the major differences of the various integration architectures from section 3.2.

Integration Broker	**Service-Oriented**	**Event-Driven**
Centralized command and control	Applications publish their functionality	Applications react to external, relevant events
Central dispatching of requests and/or replies	Contract-based behaviour	Internal handling of events
Developers decide on communication scheme	Request/Reply communication	One-way "FYI" communication
Not well scaleable	Well scaleable; runtime binding possible	Well scaleable; events typically small data-packets
Style support depends on implementation	Best suited for Composite Application Style	Best suited for Multi-Step Process Style
	Multi-Step Process Style well supported	Data Consistency Style well supported
	Overkill for Data Consistency Style	Needs customization for Composite Application Style
Developers decide on communication scheme	Sequential flow typical	Parallel flow typical
Typical implementation by means of messaging (Message Broker pattern, [Hoh03])	Typical implementation by means of Remote Procedure Invocation*	Typical implementation by means of messaging or even file transfer

Table 2: The Integration Architectures Compared

*: Note that Remote Procedure Invocation is a common denominator for C/C++ style RPC, Java RMI, but also Web Services. Functionality is being published and remotely invoked. The typical underlying technology is some form of messaging, but it can often be abstracted from.

As can be seen, the architectures do not exclude one another. At some points, they can both be used and the decision will be based on external factors such as present experience with the development teams assigned to the project. The Integration Broker Architecture is an older design, but can be adapted to many situations. Services and Events are newer architectures and will often be more useful, especially since they themselves carry a lot of concepts with them.

4. Integration Issues

While the previous chapters were surveys of different existing fields of interest in application integration, this chapter will answer the research question itself: what are the properties of an application that define its ease of integration? In the first section, integration will be split up in domains: different areas of interest while searching for these properties. Thereafter, a model will be put forth which depicts those concerns and issues that play a role in defining an application's ease of integration: the answer to the research question. However, not all properties are of equal importance in all integration projects; the style of integration (as explained in section 2.2) shifts the focus from one series of issues to another. To illustrate this, the last section of this chapter puts forth a priority model for each of these styles.

4.1 The domains

Integration was defined in chapter 1 as "coupling the data and/or processes of applications which were independently built, in order to create additional value". In other words, a number of source components are being linked in order to create a new system, one that offers additional value. These links would be based on a certain integration style (section 2.2), depending on the value that is being aimed for. In a simple picture, this would look as follows:

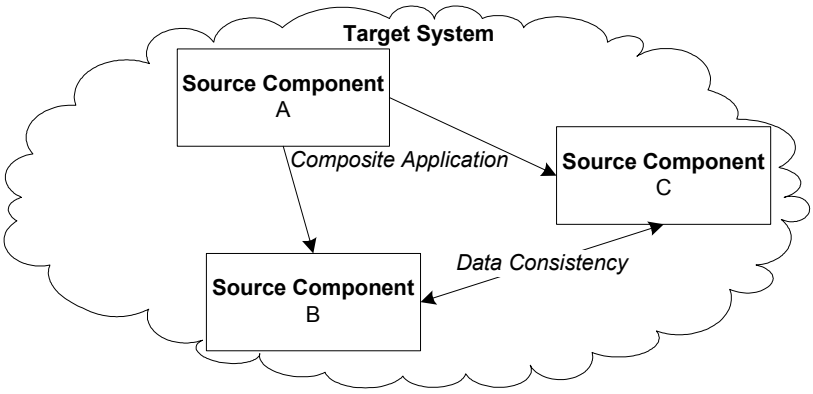

Figure 14: Abstract Application Integration

The target system is a vision for the future: the situation that is aimed for by coupling the source components, because building this future situation is done by coupling existing systems: source components from the point of view from the integration architect. These source components are the "applications" being referred to by the research question. The properties of these source components are therefore the area of interest in this thesis. However, abstracting from the target system is not possible: the properties of the source components cannot be evaluated unless compared to the *need:* the source component must be placed somewhere within the target system and linked to other source components. A comparison of the component's properties is crucial in an integration project, both a comparison with the need of the envisioned target system as well as a comparison with other source components to measure the alignment between these. The former is important to see whether or not the source component will actually fit in the big picture, the latter is required to evaluate the amount of work that will be put into linking the source components themselves.

Two important domains can be identified: the target system and the source components. One could add other, more external domains such as human resources, project management and constraints to this picture. However, the two domains that were identified here are vital in answering the research question, whereas these other possibilities are duly noted as having an effect on the integration project, but play no part in the *properties of applications* being researched here. The next section will explore both the domain of the target system and the domain of the source component, and

explain which properties in both of these are important, measuring both the source components' fit and alignment in the target system.

4.2 Concerns and Issues

This section explains just what properties an architect should look for in an application that is to be integrated. Figure 15 depicts the model that was developed throughout the research done for this thesis.

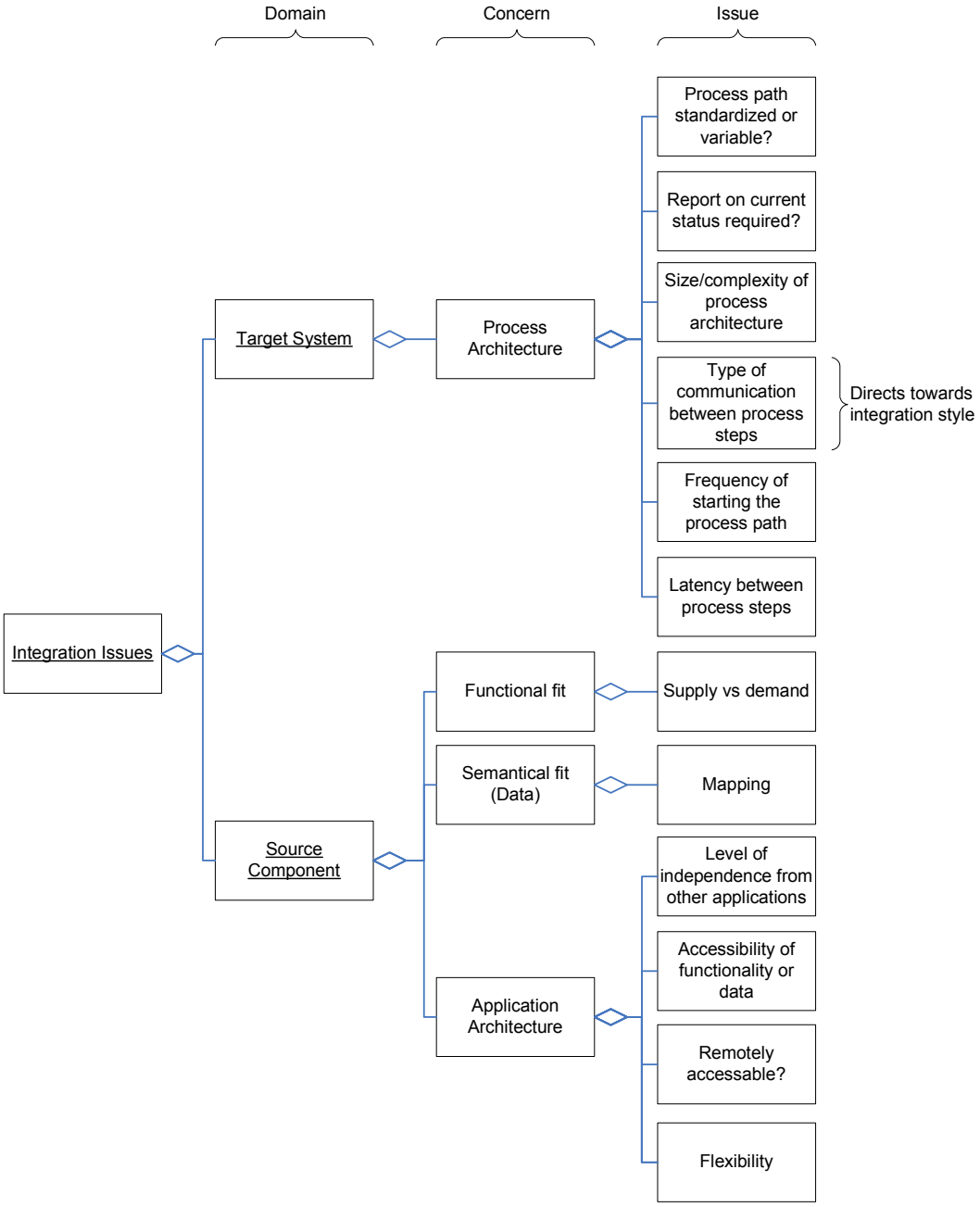

Figure 15: Integration Issues

The first refinement (branches in figure 15) is that in two domains: the domain of the target system and the domain of the source component. The existing applications, or source components, will be placed within the target system, so the architect can get an idea of the situation that is being aimed

for, before being able to evaluate an application's fit therein. Therefore, the first domain that will be explored is the target system in section 4.2.1. Thereafter, the source component domain will be explored in section 4.2.2. For each of these domains, the relevant concerns will be explained. These are areas of interest, which one can evaluate more precisely, by focusing on more low-level issues. Issues are questions or items that refer to (or inquire about) the properties that are of importance in evaluating an application's ease of integration.

4.2.1 The Target System Domain

As was explained before, an application's ease of integration cannot be evaluated by looking at it in isolation. Upon integrating an application, there is the added value that is being aimed for (as can be derived from the definition put forth in chapter 1). This added value can be made explicit by modelling the future, integrated solution in a process architecture: the future process model, combined with the boundaries of existing source components and the communication required between them. Subsequently, each of the applications that are to be integrated can be mapped onto this architecture, to see where and how well it fits in with the future situation. If an application does not fit well, a lot of work has to be done in order to put it into place after all. This is exactly what the research question is after: finding the properties that define an application's ease of integration.

Therefore, the one vital concern in the target system domain is that of the future process architecture. This architecture should depict the future process model, the one that should exist after the source components have been linked together. Six issues have been identified that play an important role in pre-evaluating an integration project, and each of these is named in figure 15.

The first of these issues is whether the process path is standardized or highly variable. If the process architecture depicts a spaghetti-like mesh of possible routes, maintaining control of the target system will not be easy and the integration infrastructure will suffer an additional burden because of the required level of communication. If the path, on the other hand, is rather standardized, each source component can be made (partially) responsible for maintaining control, or possibly even the events that traverse the path themselves can take this task upon them, basically guiding themselves through the different processes. If this isn't possible, due to the great variability in process paths, it makes sense to introduce a component to the target system whose responsibility is to calculate the appropriate process paths and direct the events through these. The addition of such a component results in additional communication as well, which is another burden to the project. In short, the level of variability in the possible process paths indicates the need for a separate business process manager or workflow manager component.

Another reason for deploying such a component might be the need for status reports, the second issue in figure 15. After integrating different source components and thus creating a larger (target) system, the result is no longer a typical simple straight-through application. Different previously existing applications within the integrated whole are now working together to complete a task, one that is likely to be split up in different responsibilities, namely those that were previously distributed between separate functional applications. It is conceivable that there exists the need for status reports: how far has a command or an event been processed within the target system? This can be a tricky question if the source components have been linked, and both the principles of encapsulation and separation of concerns have been put in effect. Especially in the multi-step process style, where an event "proceeds in an automated fashion through its lifecycle, potentially involving dozens of steps in various locations" (see section 2.2.2) it will be hard to check up on an event's location in order to get a status report. Introducing a business process manager component will allow that component to keep track of each command or event currently being processed, thus enabling the possibility of acquiring current status reports.

The third issue, the size/complexity of the process architecture, is easily understandable. The larger and more complex the process architecture is, the harder it will be to find fitting source components and to build an adequate integration infrastructure. This issue is also reflected in the variability level of the process path, the first issue. Essentially, linking two personnel

administration databases is easier than coupling a world-wide network of book vendors and shipping companies.

The fourth issue directs the architect towards one of the integration styles from section 2.2, or possibly a combination thereof. The issue at hand is the type of communication that follows from the process architecture. If processes are modelled to have two-way communication (or request-reply pairs), the most likely candidate style is the composite application. If the cooperation between source components consists of the dispatching of events that are to be processed by the next step in the path, a multi-step process style will most likely fit best. Further interpretation of the process architecture will reveal whether or not there is a simpler underlying goal: making redundant data in multiple systems agree. In this case, the events being dispatched over the communication lines may be data mutations, which are to be heeded by other data repositories. In short, the integration style that fits best can be inferred from the process architecture. Remember, though, that an integration project does not always consist out of one coupling, and one style; a combination of different styles is possible. However, for each coupling (or communication line in the process architecture) we can interpret an associated style. In section 4.3, a priority scheme is put forth for each of the three integration styles; it lists the most important issues that define the alignment (essentially the ease of integration) of two applications, given the style of the coupling between them.

The final two issues in this domain are both aimed at measuring the use intensity of the target system. The first of these is the frequency of starting the process path as modelled in the process architecture. For each of the styles, this refers to a different concept: mutations for data consistency, events for multi-step processes and service invocations for composite applications. It is an important measure, which will help in deciding what solution to deploy. Some are more capable of dealing with higher frequency communication than others (messaging vs. file transfer and an EDA vs. a broker, for instance). A full-fledged messaging system, however, is harder to build than a file transfer system, so the issue of use intensity in the form of frequency of use is definitely an important one.

The second issue measuring the use intensity of the target system, the last issue in this domain, is that of the latency between the process steps or components in the process architecture. Again, for each of the styles this translates into a different concept: how “outdated” data can be in the data consistency style, how long a component may take to process an event in a multi-step process and how much time there should be between making a request and receiving a reply in a composite application. The issue is most important in the data consistency style (as is discussed in 4.3: the styles and issue priorities) because there it directly influences the integration technique (solution) to be used. A file transfer will hardly suffice for a composite application, but it is a viable option in the case of data consistency where mutations may be shared on a twenty-four hour basis.

This section has explained what factors define the ease of integration in the target system domain. It has done so by stating the various issues that play a role when it comes to the concern of the future process architecture. This concern is the vital one in this domain, because it will be used as a basis to assess the fit of a source component when it is to be placed in the target system. Such a fit defines the amount of work that is to be done in order to integrate a component: its ease of integration. The next section discusses the concerns and issues that are of importance in the domain of the source component when assessing its fit in the integrated target system.

4.2.2 The Source Component Domain

The second of the two domains is that of the source component. The definition of integration in chapter 1 refers to the coupling of “applications, which were independently built”. The research question then inquires about the important properties thereof, in order to measure the ease of integration. This section is dedicated to explaining the issues that were found in the domain of the source component, in other words the properties of an application that define its ease of integration. Remember that an integration project is the coupling of multiple applications; this

section explores the domain of the source component, so the issues discussed here can result in different considerations for each of these source components.

Functional Fit
The fact that ease of integration cannot be measured as such, but requires a level of comparison between different influencing factors (domains) has been explained before. The functional fit of an application is an excellent example. For this concern, there is basically one issue that has to be answered: does the source component offer the functionality that is required by the target system? In other words, this entire concern is a matter of comparing the supply with the demand: the source application's functionality with the target system's need as inferred from its process architecture. If the target system in fact demands more than the source component's current functionality, it will have to be expanded, introducing both additional risk and workload. The opposite can also be the case: two separate source components can have an overlap in their functionality. This requires careful consideration: can such a situation be allowed to exist or should measures be taken? In the latter case one could imagine having to modify the current source components or introducing a separate component to control the effects of said overlap. Most often, however, integration projects are undertaken with the different source components in mind: their combination has to offer additional value. A functional gap is therefore more likely than a hindering overlap, introducing the need for the creation of additional functionality. In short, a perfect functional fit between supply and demand would be best, otherwise only more work and risk are introduced.

Semantical Fit
Making source components work together means having them communicate. The need for communication indicates the need for a common ground of understanding. This common ground should be on the level of the mutual data models, because these define the point of view each source component has on reality; applications work with models (abstractions) of reality. An application's data model defines what it understands and how it reasons. If two independently built applications are to be coupled, the common ground between their data models defines their semantical fit. An example used in the first chapter on data semantics was that one application might see a client to be a company, which happens to have a contact person; another may see a client to be a contact person, which happens to work for a certain company. These differences will have to be overcome by mapping one data model onto the other.

The semantical fit directly influences the amount of work that has to be done in order to have multiple applications communicate about the same concepts. However, the required "size" of the common ground, in other words the level of semantical fit, depends on the integration style that will be deployed. For instance, in the data consistency style the fit has to be near perfect in order to facilitate an easy integration of the different data sets involved. This will be further explained in section 4.3, where the different priorities are discussed for each of the three integration styles.

Application Architecture
After having attempted to fit the source component in the target system, alongside its compatriots, one should have a fairly good idea of the amount of work that is needed to make it fit in. There are, however, also some issues under the concern of application architecture, which play an important role in defining an application's ease of integration. So far, the focus has been on properties of an application that play an external role; issues that only meant something if compared to others. It is time to turn to the properties specific to one application, to one source component. Four issues have been identified, each of which will be addressed next.

The first of the four issues under the concern of application architecture is that of the level of the application's independence from others. The analogy between application and source component as used throughout this thesis illustrates the integration architect's need to see the applications that are to be integrated as segregated entities. In the ideal situation, an integration project would consist of choosing the appropriate components and plugging them onto a pre-defined infrastructure in order to have them work together (imagine building a PC by plugging the appropriate cards onto the motherboard). Aside from building that infrastructure, another problem may be that an application is not as isolated as the architect would want it to be; it may depend on other applications in order to function correctly. This may be the result of a previous integration

project, or because it was designed as such in the first place. A personnel administration package may include a client-side user interface application and a separate server-side data management application. If the server-side is to be integrated with another personnel administration (as might be the case with a company merger), the client application might still be expected to function correctly. This is a very low-level example, but dependencies may exist on a larger scale as well. These dependencies introduce constraints: both in the possibilities of altering the source components and in the fact that existing dependencies are likely to continue to exist. A fully isolated component is easier to modify and "plug in" than a business critical core component with many hundreds of employees depending on it through different applications.

The second issue is that of accessing (reaching) the functionality or data of a source component. Under the previous two concerns of functional and semantical fit, the functionality and data model of a source component were compared to the functional need and other (possibly different) data models respectively. Having decided to use an application in an integrated target system, after determining its fit (both functional and semantical) in the integrated whole, it is time to unlock the application. This unlocking refers to revealing the application's functionality and/or data for use in the integrated target system, depending on which of the two is going to be integrated. The amount of work required to unlock either is therefore directly related to the existing level of accessibility. In a monolithic application, the access to either functionality or data is more of a problem than in an application built by means of a layered architecture. Figure 16 depicts such a situation, with an illustration of the different styles and their "point of entry" in the source component. Since not all applications were built with such a separation of concerns in mind, the level of actual accessibility is an important property when it comes to the ease of integration.

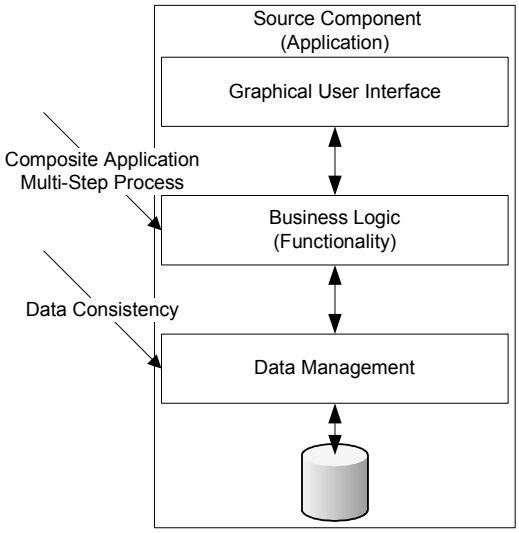

Figure 16: A Three-Tier Layered Architecture

Accessing the appropriate part of a source component is one thing, remotely accessing it is yet another. Therefore, the third issue under the concern of application architecture is whether or not a source component is *remotely* accessible. Most, if not all, integrated systems are distributed systems as well: its various components run on different physical systems. Therefore, the connection between these components not only consists of possible messaging and transformation schemes, but also some form of network communication such as TCP or RMI. If a source component is already remotely accessible because such a mechanism is in place, it saves the architect a big burden. In a typical Service-Oriented Architecture (section 3.2.2), an application is designed to remotely communicate with others: its interfaces often already incorporate the necessary functionality. A good example is Web Services and the SOAP communication paradigm. The other end of the spectrum would be a monolithic application, built to run on one system without being accessed from anywhere but the system's keyboard; such a system is not built to communicate over a network at all.

The final issue of a source component's application architecture is its flexibility or its modifiability. This is a non-functional requirement often reached by implementing a layered architecture (figure 16), black-box principles and separation of concerns. Application integration is often achieved by modifying existing components: expanding them, sometimes tweaking them to make them fit in with the integrated system. This can be on a functional or on a semantical level, depending on the need for modification, which results from the functional and semantical fit respectively, as discussed before. Flexibility is a term used here to address the difficulty of making such modifications: it's not only the amount thereof, but more importantly the effort it will take to achieve them, that matters in defining an application's ease of integration.

The concerns and issues discussed in this section are not always equally important. Throughout the research project, it became clear that the different styles of integration also have a different focus on these issues: some weigh more heavily than others. The next section discusses each of the three integration styles and the top three of issues that was identified.

4.3 The Integration Styles and Issue Priorities

Knowing what issues can be considered enabling or disabling, prior to commencing an actual integration project by creating the infrastructure and communication endpoints, is definitely an advantage. However, as was stated when the "type of communication between process steps" issue was discussed in the domain of the target system (section 4.2.1); some of these properties weigh more heavily than others, depending on the style of integration. Section 2.2 explained the three different styles, each with a different integration goal being worked towards. These three styles require different technical solutions and thus a different focus on the issues that were discussed in section 4.2.

For instance, compare the data consistency with the composite application style: the former is aimed at integrating data, the latter at integrating functionality. Consequently, the former will also be less concerned with the issue of functional fit than the latter: functionality hardly plays a role. Therefore, the ease of integrating two applications in a data consistency style does not depend on the functional fit of these. Instead, another focus on certain issues should be applied. The same goes for the other two styles: some issues play more important roles than others in defining the ease of integration. This will be explained in this section by going through each style and explaining a focus, aimed at the most important issues within that particular style. Figure 17 depicts these different top issues per style. Sections 4.3.1 through 4.3.3 discuss each of the three styles respectively.

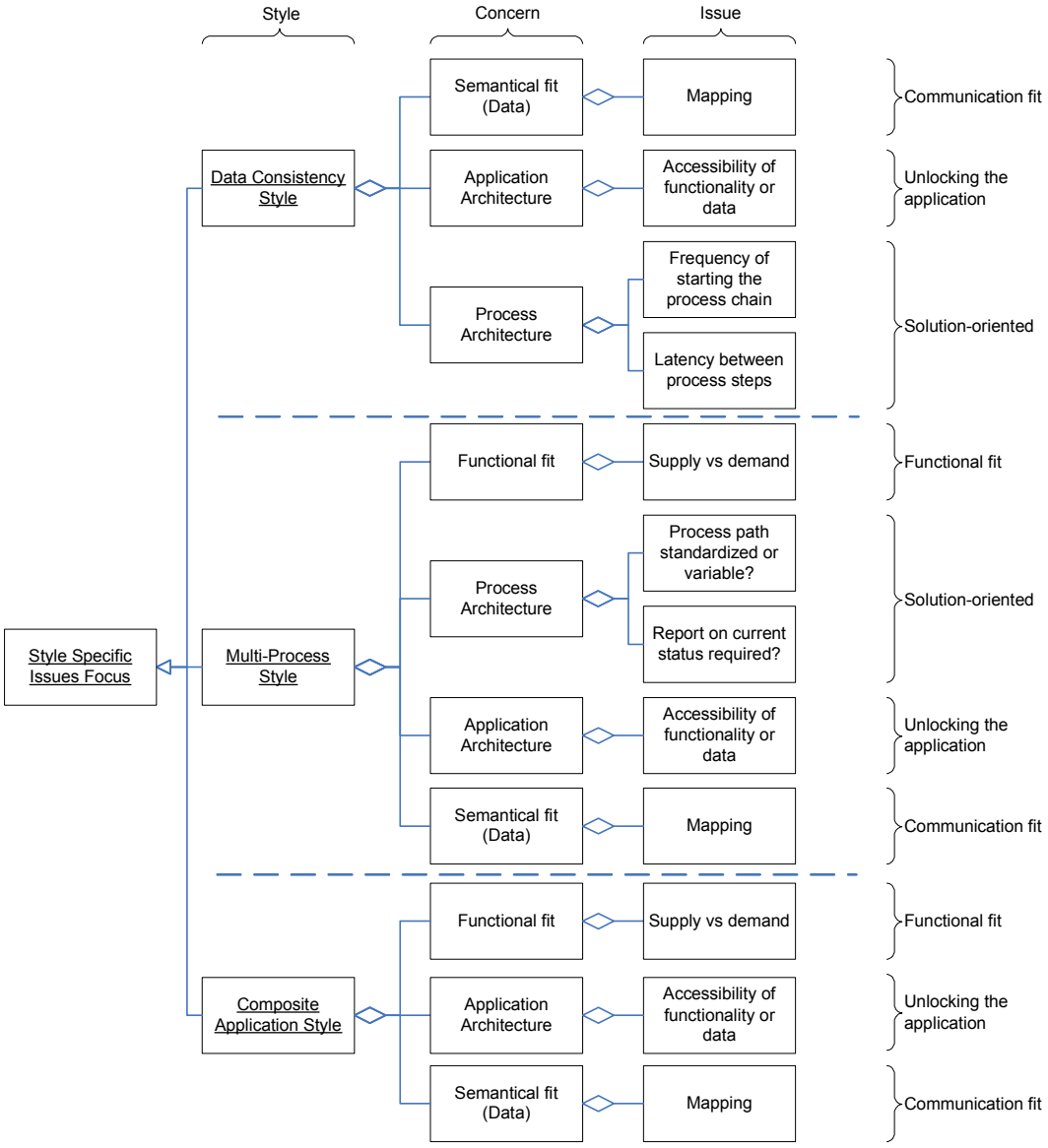

Figure 17: Issue Priorities

4.3.1 Data Consistency Issue Focus

Recall from section 2.2.1 that the general, unifying goal of all data consistency projects is for redundant data in multiple systems to agree [Fri04]. However, before a series of applications can agree on the *status* of their mutual data, they should first agree on the *semantics* thereof. This is why the concern of semantical fit (in the form of data model mapping) between applications is the number one issue when it comes to integration through data consistency. In this style, the mapping of the different data models (semantically) has to be near perfect: the creation of one (virtual) data repository is the goal. The "communication fit" as it is named in figure 17 in a more solution-oriented term, should be almost absolute.

Just the semantical fit, however, is not enough. Being able to get to the data is also an important issue, in other words: unlocking the various source components. Data consistency means "catching" one application's data mutations and subsequently forwarding this to another in order to make sure the update is propagated throughout (or at least "seen by") the entire integrated

system. Therefore, the accessibility of the source components' data is the second issue in this style's focus.

The third important issue is solution-oriented: what technical solution will fit the need? Here, the important consideration is a combination of the frequency of starting the process chain and the latency between the process steps: in the case of the data consistency style this translates into the frequency of relevant mutations and the time constraint of sharing these. In other words: will a file transfer each twenty-four hours suffice or is a "live" connection necessary?

This concludes what was found to be the top three of issues in determining the ease of integration by means of a data consistency style. One important issue was left out: functional fit. This is entirely due to the very nature of the data consistency style: no functionality is being shared; the style is solely aimed at sharing a consistent view on the current data. The following section continues with explaining the focus of issues when it comes to the multi-step process integration style.

4.3.2 Multi-Step Process Issue Focus

Section 2.2.2 defined the multi-step process style as follows: "the premise of the multi-step process integration style is that a transaction is entered only once. Thereafter, it proceeds in an automated fashion through its lifecycle, potentially involving dozens of steps in various locations, without re-keying the data." The most important items in this definition are the lifecycle and the "dozens of steps" mentioned. It refers to the process architecture for this new and automated situation, where the lifecycle and steps can be translated into the process model and its steps. Each of these steps should map onto a source component, where the appropriate processing is invoked. In other words, the supply and demand of functionality and the fit thereof is the one most important issue when it comes to integration in a multi-step process style. If this functional fit does not exist, or at least up to a certain level, the integration project is futile: no added value will be gained.

If the source components do map onto the various process steps and are thus appropriate for use in an integrated target system, the next concern is a solution-oriented one: namely the question of whether or not a separate business process manager or workflow manager component is required. This conclusion can be drawn from two issues, both under the concern of (target) process architecture: the variability of the process path and the need for status reports. Both of these issues and why they direct towards a possible management component were explained in section 4.2.1. Clearly, an integration solution without the need for a management component is less complicated than one where the need for such a component does exist.

Another part of the solution is unlocking the various source components for use in the target system. In the case of a multi-step process this means gaining access to the functionality of these various components: a certain process is to be activated after all. Therefore, the third concern in the multi-step process style focus is that of application architecture, in the form of the issue of access to a source component's functionality.

A fourth concern in this focus is the semantical fit. The definition of this style refers to "entering a transaction" and "without re-keying the data". This implies the need for a common view on such a transaction, for the correct interpretation of events that are being communicated. This would be the communication fit that was number one in the data consistency style. Within this style, however, the semantical fit will have to be but a margin of the all-but-perfect fit in the data consistency style: only a common ground for correct communication between source components is required in a multi-step process style. In other words, not a complete data model mapping is necessary, only the mapping of those parts of the model that are used to communicate data. The next section explains the last of the styles and its top three of issues: the composite application integration style.

4.3.3 Composite Application Issue Focus

The composite application style was referred to in section 2.2.3 as: “Composite Applications involve immediate interactions among two or more semiautonomous applications, working in concert to execute a single step in a business process.” The key notion here is the fact that independently built applications are now supposed to “work in concert”. This immediately suggests the need for a close functional fit, which is the first concern within the focus for this style. And indeed, integrating two source components and creating a “super component” that combines their respective functionality requires a close functional fit: the entire basis of an integration project in this style is the combination of functionality.

However, a functional fit alone is not enough for applications to “work in concert”; the functionality of a source component will have to be accessed in order to integrate it with other components and their functionality. The ideal situation would be much like the Service-Oriented Architecture as discussed in section 3.2.2. In such a situation applications publish their functionality for use by other applications by means of interface contracts. Functionality is seen as a service, which clients can make use of. If this situation does not exist as such, there is still the hope for something along the lines of a layered architecture (see figure 16), in which case there might still be the need to add a layer to enable a service-like remote request-reply mechanism. In the worst case, a great modification is necessary in order to be able to access the source component’s functionality.

Another important part of the definition and the number three concern within the focus of this style is the interaction between applications. Of course, there has to be some form of “common ground” between the different components in a composite application on the level of their data models in order to have them cooperate. If this does not exist, a common understanding about the concepts and items being communicated cannot be achieved. Therefore, the semantical fit is definitely an important concern for this style, even though it’s a notch down from the semantical fit required in the data consistency style, where an all but complete mapping is required.

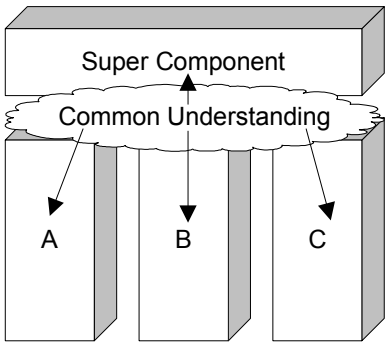

Figure 18: Common Understanding in a Composite Application

In a typical composite application, the super component would refer to different subcomponents for different functional needs. Recall the example from section 2.2.3, the on-board navigation system. It referred to one subcomponent for roadmap information, with which it will likely communicate in terms such as zip codes. On the other hand it referred to a traffic information system, which will communicate in other terms such as accidents and slow-driving traffic. There is also an overlap, possibly in the form of road indicators (E12, N256, etc) which forms the common understanding even between the different subcomponents.

This concludes the last of the styles and the discussion of its most influential issues. The concerns and issues explained in this chapter, however, are still rather abstract. The next chapter will therefore discuss two cases, which were researched at PinkRoccade, in order to illustrate a more practical view on the concerns and issues that were discovered.

5. Integration in Practice

This thesis puts forth, in chapter four, a theory on pre-evaluating the difficulty of an integration project, based on several properties, separated into two domains. In order to come to this theory, both practical and theoretical knowledge were combined. At PinkRoccade, two cases of integration projects were examined, and an integration architect was available for explanation and deliberation. Through his knowledge, not only the two cases as described in this chapter have served as a starting point for the model of figure 15 on concerns and issues, but many others were drawn into the discussions. The resulting knowledge and diagrams from these discussions were used to take a more top-down, abstract view in order to categorize the different concerns and come to a theoretical model. The result of these iterations has been explained in the previous chapter.

While the cases that were used as a starting point for the research are more than one step away from the resulting theory, it might still be interesting to go through them. This chapter explains about *VTS* and *Polis*: the two cases that were used as a starting point. These will now serve to illustrate the point that was made in the previous chapter.

PinkRoccade Public is a division of PinkRoccade aimed at serving the governmental sector's IT needs. Within this division, there is the department of *SOZ*, that of social securities. Historically, this was once part of the *GAK*, one of the social security agencies in the Netherlands. This is where the case studies have taken place, and this is therefore the realm thereof.

Recently, all of the five different social security agencies in the Netherlands (*GAK, GUO, Cadans, Bouwnijverheid* and *USZO*) have merged into one, called the *UWV*. Of course, all of these different agencies had been using their own IT systems to keep track of employers, employees and social security payments. The different social securities (unemployment benefits, health care etc) were managed by different applications. Separately from these applications, each agency had its own central data storage. All of these systems were not all built by the same supplier, so even though some serve the same broad purpose, there are still many differences. The *UWV* commissioned two projects to shut each of these separate systems down over time, and replace them with central ones, administrated by the *UWV* itself. These projects are called *Convergentie* (convergence) and *Polis*. The first consists of many smaller projects, aimed at reducing the different applications in use per social security. *VTS* is one of the projects within *Convergentie*, aimed at shutting down four of the five previously existing unemployment benefits systems, and use the fifth throughout the *UWV*. *VTS* will be explained in section 5.1. The second project, *Polis*, is aimed at merging all the data stores currently in use, and creating one central registry for all of the *UWV*. *Polis* will be discussed in section 5.2. Figure 19 is a simplified representation of the merger of the different former social security agencies into the *UWV*.

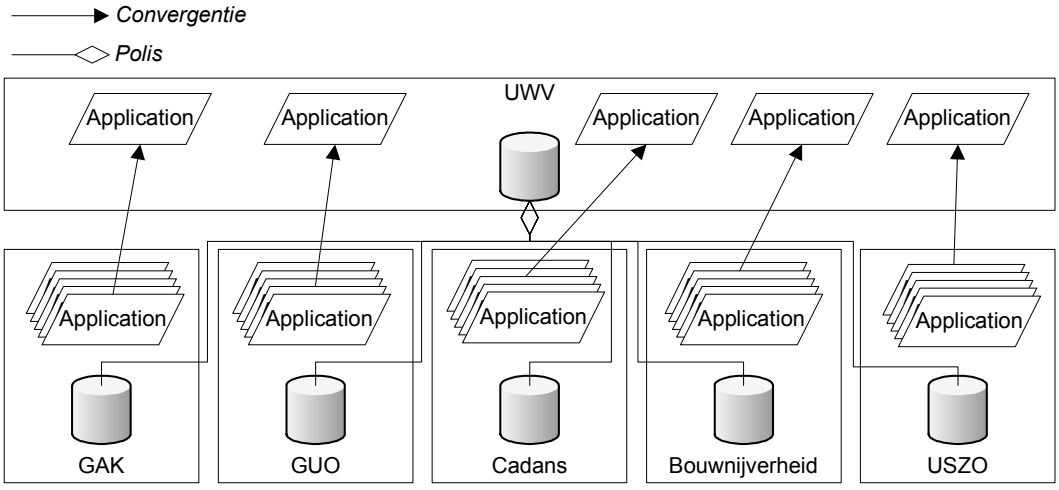

Figure 19: The Merger into *UWV*

5.1 VTS

VTS stands for the Dutch "*vertaalservice*" or "translation service". This fits in with the nature of the project: a translation component between the different employer/employee registries of former social security agencies, in order to have one unemployment benefits application make use of all of these.

This section will explore *VTS* by first explaining about the project itself, before discussing the focus of the appropriate style as put forth in section 4.3

5.1.1 What is VTS?

VTS was conceived to reduce the number of unemployment benefits applications running at the five different former social security agencies. Three different agencies were appointed to have their systems merged so far: *GAK, GUO* and *Cadans.* The client that would be used to consult all of these systems' employer/employee data is *DSC WW* or Target System Complex WW (WW stands for the Dutch unemployment benefits), previously in use by *GAK* only. Figure 20 depicts the functional architecture of such a system.

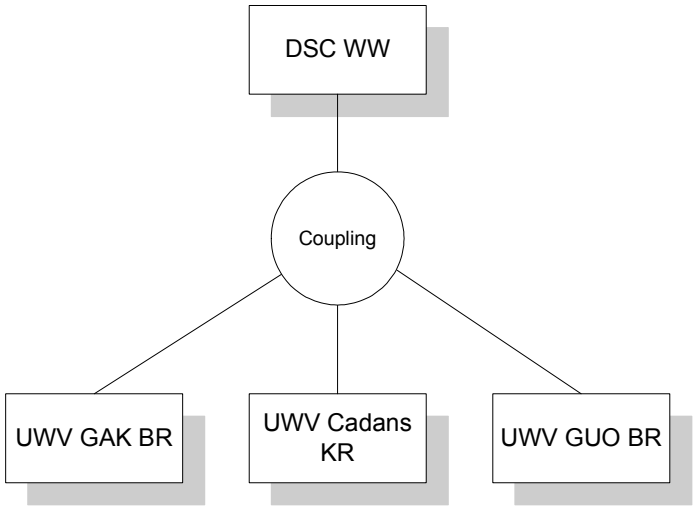

Figure 20: Functional Architecture of *VTS*

Throughout this chapter, authentic Dutch figures from the system documentation were translated partially into English; the original names were left in place, to prevent confusion. In this case, *BR* stands for Central (*Basis*) Registration and *KR* for Core Registration. These last two are basically the same kind of systems: the registration of employer/employee data.

What the *DSC WW* component does, basically, is querying the different registration systems of the three separate social security agencies and combine their data into one representation for the user. On the one hand, this is a typical Composite Application (section 2.2.3), where the *DSC WW* forms the super component, making use of the functionality from the three different registration systems. On the other hand, however, there is a discerning property from the typical composite application: the super application does not make use of different services in order to combine them; instead it makes use of similar, overlapping query functions on a series of data sets. By doing so, it gets the most complete answer to its queries. Depending on a certain variable such as the employer, the system could in some cases deduct the fact that the information will (only) be available in one of the three systems. In such a case, the query will be routed to the appropriate system only. This is depicted in the target system architecture of figure 21.

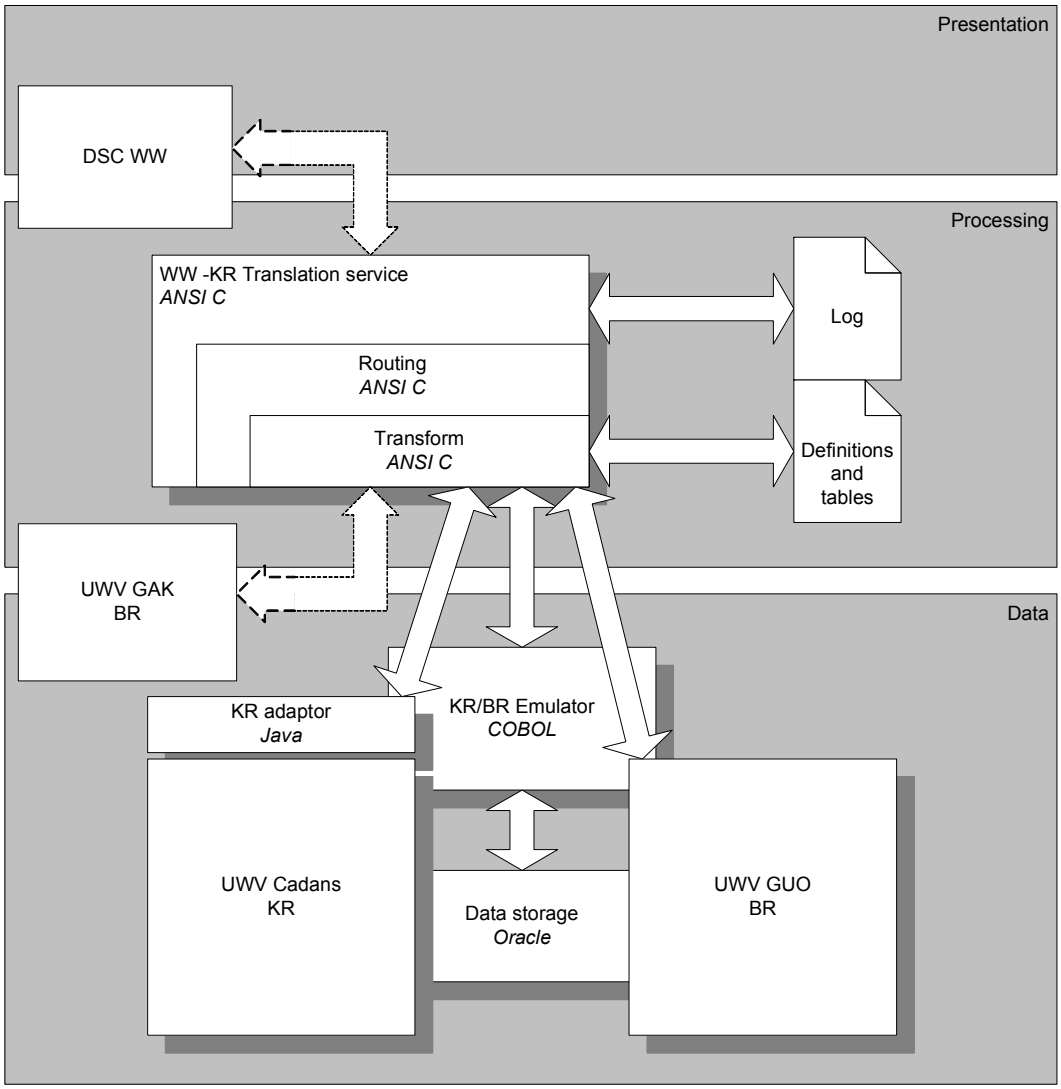

Figure 21: Software Architecture of *VTS*

In this figure, the three (grey) layers are called "presentation", "processing" and "data" from top to bottom. As can be seen here, the processing is done by the *VTS* component: it routes and transforms queries, and subsequently combines and transforms the resulting data. The existing *GAK, Cadans* and *GUO* components reside in the data layer, serving as repositories only.

Remember that the *VTS* project is part of the larger *Convergentie* project: replacing four of the five different existing applications within the realm of one social security with the fifth, which was chosen to remain. *VTS* does this for the unemployment benefits.

5.1.2 Concerns/Issues

In the *VTS* project, the focus was on three concerns, in line with the theory of section 4.3.3 on Composite Application Issue Focus. These were the functional fit, the application architecture and the semantical fit. Each of these will be discussed next.

Functional Fit – Supply vs. Demand
As said before, the *VTS* project was not such a typical composite application in the sense of combining functionality. The functionality was actually very much the same for each of the source components involved. The added value when combining these components was in a more complete view on the available data. The functional demand, in this case, was that of supplying the

required data. The supply was indeed just that. This is due to the very nature of *VTS*: it was conceived with the functional fit in mind from the very start. The supply/demand combination will be more interesting if there is a choice between different components offering slightly different functionality or services.

Application Architecture – Accessibility of Functionality or Data
In the case of a composite application, the required access is to the functionality of each of the source components, not directly the data. In this case, each of the source components was originally developed to serve as a central point of access to the employer/employee data, by means of a large number of clients distributed throughout different locations. Therefore, the original design closely fitted the new situation, only this time it was not the UI that accessed the data directly, but instead the *VTS* component. In order to overcome the major differences between *Cadans* and the *VTS* component, a local adapter was fitted on the old *Cadans* system. This meant that *VTS* could access the *Cadans, GAK* and *GUO* systems in a similar fashion, instead of constantly making exceptions for *Cadans.*

Semantical Fit (Data) - Mapping
The previous two concerns were easily solved due to the original set-up of the source components. The semantical differences, however, were rather more important. This conclusion could even be drawn from the name of the project itself: an acronym that stands for "translation service". The majority of the documentation, and also the effort, was aimed at analyzing and bridging the differences in the data models and representation: both semantics and syntax. This may also be due to the rather unusual nature of this composite application: not a combination of dissimilar functionality, but of similar functionality in order to get a complete picture of the underlying data. This results in a necessary overlap in data models that is greater than in an average composite application (see figure 18). In the case of *VTS*, if an employee has worked for an employer that used the *GAK* system for the registration thereof, and started working for another that used the *Cadans* system, both of these relations will have to be represented in the *DSC WW* UI component, thus requiring a mapping between these two different systems' data.

In conclusion, the *VTS* system does fit in the composite application style, but is not a typical example thereof. Its functional overlap between the different source components is testament to this. An interesting result is the fact that the effort and risk no longer went into the concern of functional fit, but even more into the semantical fit. The concern of application architecture remained an important one in the form of accessibility to the source components' functionality, but due to the original setup of each of the systems this was easily overcome.

5.2 Polis

Polis is the *UWV*'s long-term planning for an integrated social security system. The term *Polis* is after the Dutch word for insurance policy: an employer's social security package in this case. The goal is to create one front-end from which an employer's total employment history can be accessed, no matter where he has worked throughout the years.

Polis is taken in steps and is, as a matter of fact, still ongoing. One of the steps taken in the creation of one central database is the migration of the old data repositories, so as to shut them down one by one. This part of the project is called *Polis Migratie*, and that is the part that has been focused on during the research, and thus will be explained here.

5.2.1 What is Polis?

In order to come to a flexible and lasting solution, the first step was to create a canonical data model, the *UWV* Data Register, or the *UGR.* This defines the data model that is to be used at the *UWV*, with all its entities, relations and attributes. In the *UGR,* the meaning of all of these is defined. Expanding on the *UGR*, a messaging standard has been defined as well, based on XML, called *UwvML*; it is based on the *UGR* data model, but expands it with both the syntax and

envelope definitions for messaging purposes. Through the *UGR* and *UwvML*, a canonical data and messaging model have been laid down as a foundation for communication internal to all *UWV* systems.

The eventual goal of *Polis* is to create a single data repository for all relevant data. However, for the time being, the *UWV* is faced with five different existing repositories, all operated by the different former social security agencies. In order to replace these, a phased shutdown of each of these has been planned. But these legacy systems cannot be shut down just like that: they each contain valuable data that cannot be lost. Therefore, one of the steps in the whole of *Polis* is the *Polis Migratie* project: the synchronization of the data from each of the legacy systems with the central *Polis* data store. First, each of the former social security agency's systems will be linked to the *Polis* project central message bus (SIP – System Integration Platform) by means of a coupling component: the CTS, or Communication and Transformation Server.

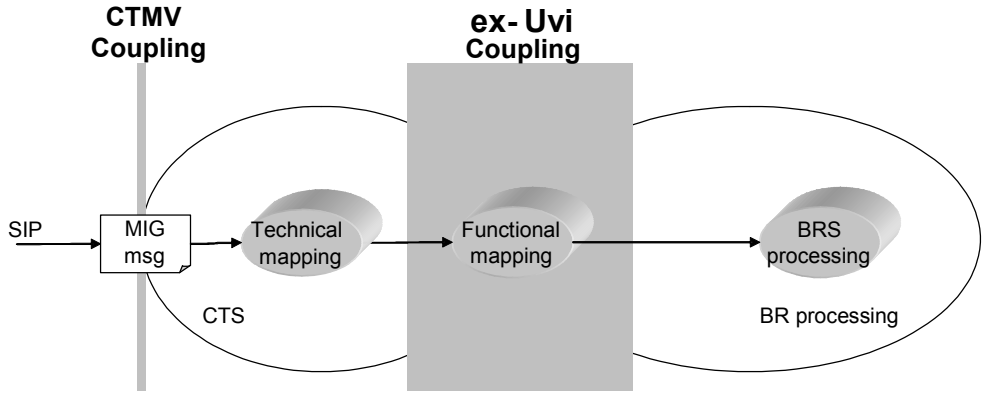

Figure 22: Coupling Legacy Systems to SIP

This CTS component will receive data mutation (MIG) messages from *Polis* through the Central Technical Migration Facility (CTMV) component: one central component in the *Polis Migratie* project, responsible for process management and routing of messages throughout the SIP. Each of the former social security agencies' systems can expect to receive these MIG-messages through its own CTS component. In fact, each of these former system development/maintenance teams was given the freedom of defining their own CTS in terms of technical and/or functional mapping from CTMV to their own system. That's why figure 22 depicts the CTMV coupling as a thin line, while the former social security agency (*ex-Uvi*) coupling between CTS and legacy system is depicted as a grey area: each *ex-Uvi* has some freedom in this area and has been asked to deliver a technical specification for their specific CTS. Figure 23 depicts the positioning of each of the CTS components next to the CTMV component.

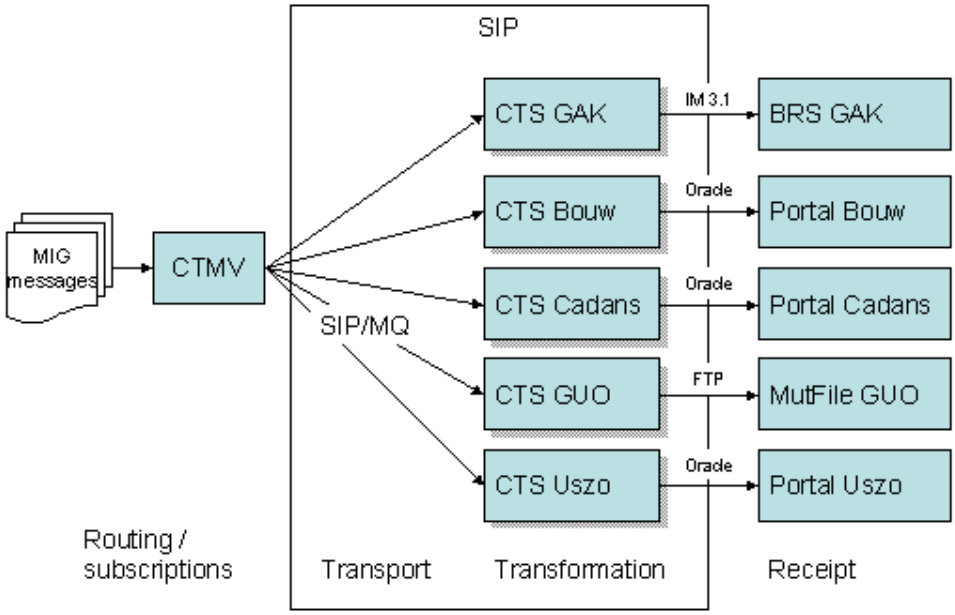

Figure 23: CTS Component Positioning

The goal of *Polis Migratie* is a data consistency integration style for each of the five *ex-Uvis* and *Polis* itself. Whenever *Polis* makes mutations on its data, it makes sure that the *ex-Uvi's* are notified. It does so through the CTMV component, which routes the MIG message (essentially a mutation notification) to the appropriate CTS. The CTS component is managed by each of the *ex-Uvis* itself, in order to create a minimal implementation for each of the five situations. The CTS component makes sure that the legacy system behind it understands and processes the mutation.

Remember that *Polis Migratie* is but a part of the entire *Polis* project. Eventually, each of the legacy systems will be shut down; its data fully migrated to the central data repository of *Polis*. The *Polis Migratie* project itself, however, has proven an interesting case in the realm of data consistency style integration.

5.2.2 Concerns/Issues

The focus of the *Polis Migratie* project has been in line with the focus as was put forth in section 4.3.1 on Data Consistency Issue Focus. The three most important concerns explained there (which can be seen in figure 17) are the semantical fit of the different source components, the accessibility of the data of each of these components and the process architecture's issues pointing towards a solution technology. Each of these will be discussed next.

Semantical Fit (Data) – Mapping
Polis is a project aimed at integrating similar data from five different source components. *UGR* and *UwvML* define the canonical messaging and data model to which all of the *UWV* will have to comply. Such a canonical data model prevents the need from a 5x4 mapping: a mapping from each legacy system's data model to the four others. Instead, it allows a mapping of each of the legacy system's data model to the canonical one, which serves as the common understanding between each of the source components. In this case, this mapping is complete: the eventual goal of *Polis* is to replace each of the five legacy systems, and no data will be lost in the process. *UwvML* is, as a matter of fact, still being tweaked at the time of writing, but it holds a great promise for being the lasting solution the *UWV* needs. In *Polis Migratie* it is put to use for a data consistency integration style for each of the five legacy systems, but it will be used more throughout the *UWV* in the future.

Application Architecture – Accessibility of Functionality or Data
The accessibility to each of the source components is, as a matter of fact, the same as in *VTS*: the original design of each of the components allowed for it to remain much the same, receiving mutations through CTMV instead of through the original channels. However, the CTMV component does not translate, as did *VTS*. Instead, an interesting approach has been taken in order to bridge the differences between *Polis* (in the form of *UwvML*) and the source components: each of the *ex-Uvis* themselves were allowed to define that bridge, thus forming the five CTS components. First and foremost, CTS was supposed to translate between *UwvML* and the local data model, but it was up to the *ex-Uvis* to include possible additional functionality. Two ends of the spectrum are *GAK* and *Bouwnijverheid*.

GAK has decided to let the CTS component make a translation that was as complete as possible, so as not to modify the existing system. Not only the XML is translated to COBOL records (as used by *GAK*), but the results are even interpreted, so as to invoke the correct processes in the original system (see figures 22 and 24).

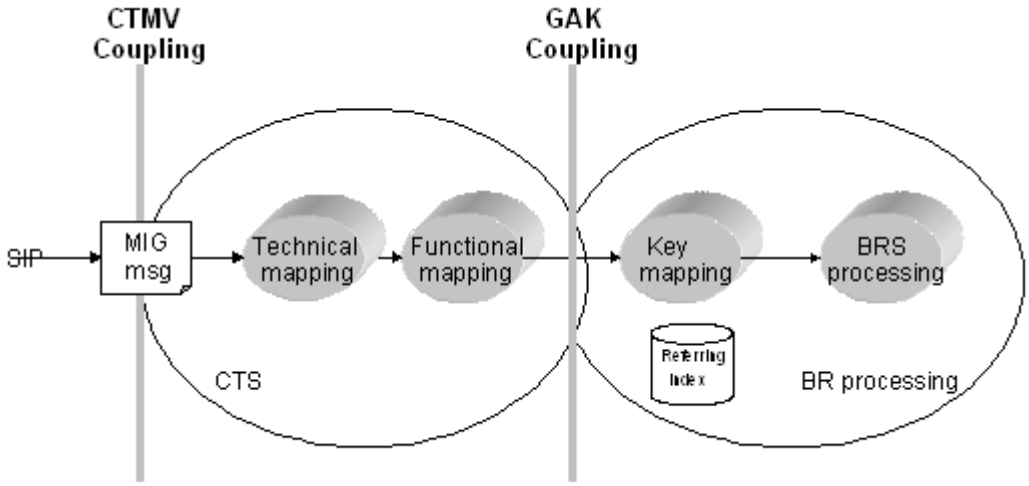

Figure 24: *GAK* Coupling

On the other hand, *Bouwnijverheid* has made the decision to let CTS translate the XML it receives into a relational Oracle structure, which is subsequently interpreted by the original system itself in order to be processed (see figures 22 and 25). This decision was mostly based on the experience at *Bouwnijverheid* with this approach.

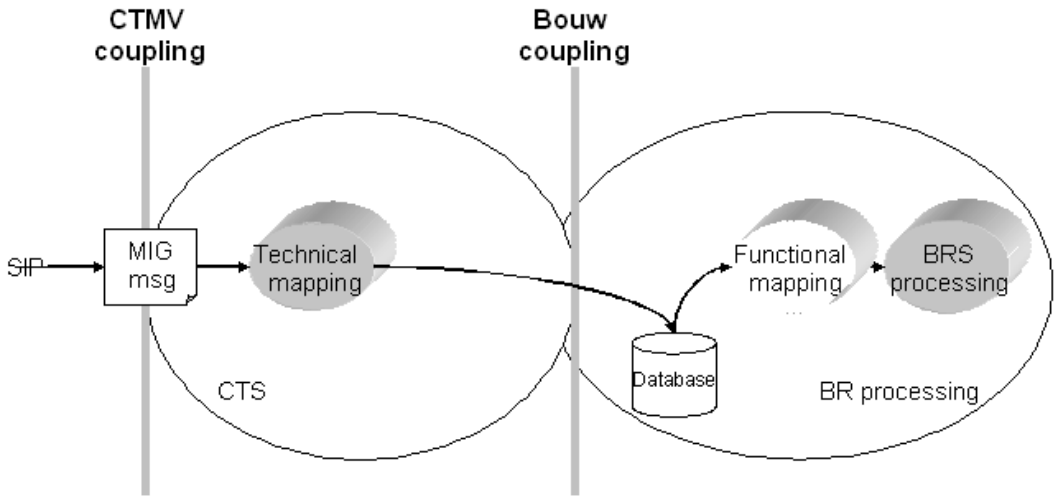

Figure 25: *Bouwnijverheid* Coupling

This approach of letting the *ex-Uvis* influence the CTS component has allowed for a range of different, yet most appropriate, solutions to exist.

Process Architecture – Frequency and Latency
This concern, as can be seen in figure 17, is solution-oriented: it directs towards a possible solution space. In this case, mutations in *Polis* were deemed important enough to be immediately processed by the appropriate legacy systems. Therefore, a messaging solution has been chosen, based on PinkRoccade's InfoMessaging MOM (Message Oriented Middleware) infrastructure. This choice, however, is also based on experience: InfoMessaging has existed for some time now, and was already in use by the legacy *GAK* system, for instance. Messaging and XML (in the form of *UwvML* in this case) is, of course, a perfect match. The decision to go for messaging was a logical one and InfoMessaging as a specific implementation thereof yet another. The "live" connection that was required in *Polis* has directed the use of InfoMessaging.

Concluding the *Polis* case, one can look back and see that a lot of time and effort have gone into defining *UwvML*, essentially the canonical data model and its messaging extensions. Also, an interesting approach has been taken in defining the coupling between legacy systems and integration platform: a buffer has been defined, in which the CTS translation component is placed. This CTS component is not defined on a project-wide basis, but for each legacy component separately instead. These two items (*UwvML* and CTS) also cover two of the three concerns within the data consistency style focus: semantical fit and accessibility to functionality or data. The third concern within the focus has guided the architects and developers towards the use of messaging, a flexible and logical decision.

6. Concluding remarks

What has been accomplished so far and what remains to be done? The main research question is often one step in a series, as is the case this time. This section describes both the scientific and practical conclusions, as well as the possibilities for future work.

6.1 Scientific Value

The research question has been answered by means of a series of concerns and issues, divided into two domains. Throughout the research, it became clear that one cannot just look at one application and say anything about its ease of integration as such. There is always another application involved when it comes to integration, and therefore a comparison is in order; one should measure the alignment of a series of source components with one another, and that results in an answer on the question of difficulty (or ease) of integration. But not just that, there are also goals involved: an integration project never has integration as such as a goal; there is always additional value to be gained. This value has to be made explicit in the form of a process architecture if one is ever going to align the source components in the correct way. Otherwise, the coupling that is built may not be the one that creates the value that was aimed for.

The diagram clearly require the architect to think of several distinct issues; they are no longer left implicit and imaginary (such as “interoperability” or “integrability”), but these have been dug out, revealing the properties that play a role. The question now is: can we build applications with the goal of integration in mind from the start? Most likely we can: SOA and component-based development are aimed at a form of “plugability” where building blocks are used to create larger components, and eventually, an application. Many of the issues that ease the process of integration are those that return in these paradigms.

In the end, these issues are all architectural forces or possibly even decisions. The GRIFFIN project [Gri05] aims to develop notations, tools and associated methods to extract, represent and use architectural knowledge that currently is not documented or represented in the system. Recall the issue of comparable documentation from the third diagram; in its description it is said that the creation of comparable architectures is bound to be one of the first steps before combining the source components with one another. Clearly, the eventual results of GRIFFIN will play an important role in that process. The issues as discussed throughout this thesis reveal those properties that should be extracted from the source components.

6.2 Practical Value

The diagram that answers the research question can be used to make architects/developers truly think about what they are getting themselves into. If the project leader or integration architect, with his staff, thinks about the target process architecture and each source component’s place therein, he will most likely fuel discussions and the exploration of alternatives when trying to map the different source components onto one another. He will also end up with an overview of the situation, where pain-points are identified; these pain-points are most likely those issues that are going to be a risk for the success of the project. Special attention should be paid to these issues, lest they are not solved properly or in time.

Another possibility is the evaluation of different available source components for one “slot” in the project. It may be the case that there is a choice between certain COTS or proprietary components. Before deciding which one to use in the project, both can be tried out on paper and evaluated to see which one will fit in better.

A third possibility is to use the issues and concerns as a starting point from where to begin deciding on a technical solution, especially for the coupling: the infrastructure. In section 4.3, some issues are already flagged as being "solution-oriented", these are the issues that will help in deciding whether or not to use file transfers or full-fledged messaging systems and whether or not to aim for a SOA or EDA paradigm. The solution space is pretty extensive, and often influenced by many more factors such as partnerships, management decisions and market workings. The basis as put forth in this thesis, however, is unlikely to change very rapidly; the properties and issues evaluated are likely to stay in place, as are proven solutions such as file transfer and messaging. SOA can be replaced, however, by newer paradigms such as Model-Driven Architecture (MDA). MDA is yet to prove itself, but it is an example of a paradigm that might just yet pop up in the solution space for integration.

6.3 Conclusions

Exploring the battlefield that is application integration is a broad issue, like many in the world of computer science. One can easily get tied up in the technological solution space of networking, messaging, wrappers, and every little component that plays a (vital) role in the infrastructure. For this thesis, however, the focus was on the source components and their eventual cooperation and coupling. Architecture was the key word, especially at the Application Architecture Layer as discussed in section 2.1. Section 2.2 then offered some insight in the possible directions such a coupling of source components can take. Three styles of integration were introduced: data consistency, multi-step process and composite application. Be mindful, however, that not all (especially large-scale) integration projects will fall into either one of the three categories; it may be the case that certain lower lever couplings fall into one of them, but the project as a whole may be a combination of styles. Chapter 3 was used to discuss a part of the solution space available to the architect, ranging from transport architectures such as file transfers to design paradigms such as Service-Oriented Design.

Chapter 4, subsequently, makes the issues and concerns that influence the difficulty of an integration project explicit by means of a diagram. It separates between two different domains: that of the target system (integrated whole) and that of the source components (applications being integrated). The issues and concerns in the diagram form the answer to the research question that was put forth in the beginning of this thesis. However, not all integration projects are focused on the same set of issues; for each of the three different styles, a focus has been put forth, explaining what issues are of prime concern when it comes to integrating a number of applications within each of the styles. Be wary that a combination of styles within an integration project will also result in a combination of focus: more issues and more concerns to look after.

Chapter 5 was used to explain a bit more about the context of the research. It illustrates how the two cases that were used as a starting point for the research fit in with the findings of chapter 4. For each of these cases, it explains something about the case, and assesses how it fits in with the appropriate integration style and its focus. An interesting piece of feedback came from the integration architect at PinkRoccade: he has witnessed how the problem of application integration has shifted from technical, solution-oriented implementation to a bigger picture: that of architecture. In the past, the question was often how to link two applications together: file transfer or messaging system? And even then, a lot of problems were going to have to be solved. Nowadays, computer science has matured. Not only are the technical possibilities better known and documented such as in the message-oriented patterns in [Hoh03], but it becomes increasingly important to carefully look at higher-level issues such as those put forth in chapter 4.

References

[Alt01] R. Altman, *What Are the Three Styles of Application Integration?*, Gartner Research Note, January 2001

[Fri04] T. Friedman, *Data Integration Technologies Support Architecture Delivery* and *Map Data-Oriented Approaches to Common Integration Patterns*, Gartner Research Notes, November 2004

[Gri05] H. van Vliet and P. Lago, *GRId For INformation about architectural knowledge,* research project at the VU Amsterdam, 2005 http://www.cs.vu.nl/~se/griffin/

[Has00] Wilhelm Hasselbring, Information System Integration. In *Communications of the ACM, Vol.43, No. 6*, June 2000

[Hoh03] G. Hohpe et al., *Enterprise Integration Patterns*, Addison-Wesley, October 2003

[Lin99] David S. Linthicum, *Enterprise Application Integration,* Addison Wesley Professional, Published: Nov 12, 1999; Copyright 2000

[Nat03] YT. Natis, R. Schulte, *Introduction to Service-Oriented Architecture* and *Most Composite Applications Will Need an Integration Layer,* Gartner Research Notes, April 2003

[Plu02] D. Plummer, *SODA Helps Developers Do Application Integration,* Gartner Research Note, November 2002

[Sch03] R. Schulte, *Use SOA for Composite Application Integration,* Gartner Research Note, April 2003

[Sch03a] R. Schulte, *The Case for Event-Driven Design,* and R. Schulte and Y. Natis, *Event-Driven Architecture Complements SOA,* Gartner Research Notes, July 2003

