Information Systems Integration Process Model

Matjaž B. Jurič, Marko Tekavc and Marjan Heričko
University of Maribor, Faculty of Electrical Engineering and Computer Science
Institute of Informatics, Smetanova 17, SI-2000 Maribor, Slovenia
matjaz.juric@uni-mb.si, http://lisa.uni-mb.si/~juric/

Keywords: integration, process, EAI, B2B, information systems

Received: June 23, 2004

Integration of information systems is a complex field where major challenges are semantic, process and technology related. Integration must be performed using methods, disciplines and activities that enable it to be effective in terms of costs and time – thus it should be supported by a well defined integration process. This article presents an information systems integration process model proposal with the goal to guarantee the quality of the integrated solution. The article focuses particularly on the integration specific disciplines: analysis of existing applications and integration design.

Povzetek: članek opisuje integracijo kompleksnih informacijskih sistemov.

1 Introduction

The growing need for the easy accessibility of information presents new challenges for information system development. This need is unlikely to be fulfilled by the separate "stand-alone" applications. Applications need to be integrated to make the information they contain available and accessible [17].

Integration is not an easy task; indeed it has become one of the most difficult problems facing enterprise application development in the last few years. The major challenges are semantic, process and technology related [16]. Information system integration or Enterprise Application Integration (EAI) as seen from the business perspective, is the competitive advantage an enterprise gets when all applications are integrated into a unified information system, capable of sharing information and supporting business workflows. From the technical perspective, EAI refers to the process of integrating different applications and data, to enable sharing of data and integration of business processes among applications without having to modify these existing applications. EAI must be performed using methods, disciplines and activities that enable it to be effective in terms of costs and time. EAI should be project oriented and should be supported by a well defined integration process.

The review of related work shows that not much has been done in the field of integration processes. In [1] the authors address the problems of EAI with ERP systems. In [2] the author addresses the problem of using middleware in integration projects. In [3] authors introduce agentified enterprise components to improve integration and cooperation. In [4] EAI is addressed from the workflow perspective. In [5] authors explain the integration of heterogeneous e-commerce applications and focus on technology questions. In [6] the use of web services for integration is discussed. In [7] the authors present a notation for modeling EAI architectures. In [8] the authors give an overview of architectures and technologies used for EAI. In [9] the component approach to EAI is presented. In [10] an XML based framework for integration is presented and in [11] a web based infrastructure is presented. None of these articles addresses the integration process. Some directives related to agile approach to integration can be found in [12], [13], [14], and [15]. They do not present the whole process however.

In this article we present the integration process model proposal which is based on the EMRIS methodology [18]. The integration process as presented in this article defines the sequence of activities to be done in a disciplined manner in order to successfully develop an integrated information system. The goal of the integration process is to guarantee the quality of the integrated solution that will satisfy the customer, will be completed on schedule, and will be within the allocated financial resources. The integration process is tightly connected to the software development process, with which it shares several disciplines. It is based on real-world experience and has been successfully used in several large-scale integration projects.

The article is organized as follows: section 2 gives an overview of the integration process, section 3 describes the analysis of existing applications, section 4 describes the integration design and section 5 gives the concluding remarks.

2 Integration Process Outline

The presented integration process is based on the following integration practices: iterative development, incremental development, prototyping, reuse, design simplification, test automation, and customer involvement.

Integration process consists of disciplines which are performed in several iterations. We focus on technical disciplines only: Requirements gathering, Analysis of existing applications, Selection of the integration infrastructure, Problem domain analysis, Integration design, Implementation, Testing, and Deployment. Figure 1 presents the integration process outline.

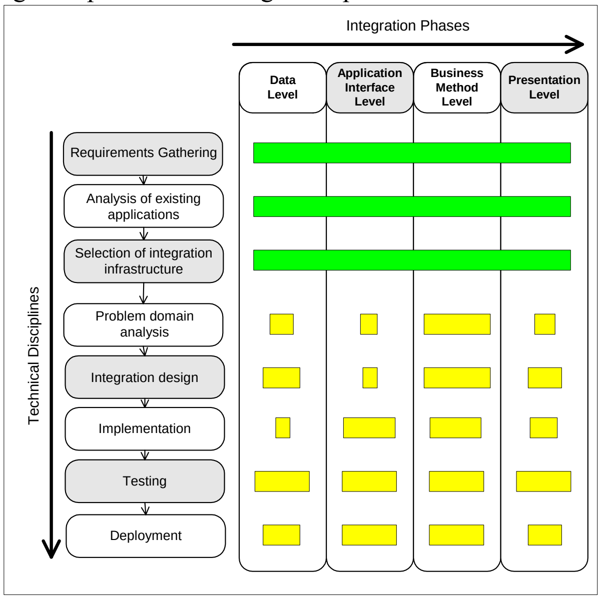

Figure 1: Integration process outline

The disciplines that are common to all phases are shown with a single box. The other disciplines are shown with separate boxes. The size of the boxes represents the approximate duration of each discipline in a certain integration phase. For example, problem domain analysis and the integration design disciplines require the most effort in business-method-level integration phase, where we have to define the global design model of the integrated information system. The least implementation effort is usually in data-level integration phase because it rarely requires changes to existing applications.

Integration is usually achieved in four phases:

- Data-level integration phase
- Application interface level integration phase
- Business-method-level integration phase
- Presentation integration phase

Each integration phase requires a lot of effort and time. Therefore, it has to be considered as a sub-project. To support iterative incremental development, each integration phase is usually broken into several iterations. Iterations enable a finer-grained control over the integration phase. Usually there are at least four iterations for each integration phase. These main iterations can however have further sub-iterations, depending on the project size and the schedule. The four main iterations for each integration phase are: inception, elaboration, construction, and transition.

Inception defines the business perspective of integration and estimates its size. We have to specify the requirements, identify all entities our system will cooperate with, and define how it will cooperate. We also have to define the milestones and the criteria for assessing the success of the integration, analyze the risks, and select the resources.

In elaboration we analyze the existing applications and get a clear understanding of what applications we have to deal with. We also analyze the problem domain, define the project plan, the basic architecture, and solve the most hazardous parts of the integration project. We also specify the requirements for the integrated information system. As we have to make architectural choices, it is very useful for us to build architectural prototypes to validate the chosen architecture. At the end of elaboration we evaluate the goals, the size of the project, and the architecture decisions, and we should once again assess the risks.

The goal of the construction is to actually implement the integration that will result in completing a certain integration phase. This part is the most time-intensive and will have the largest number of iterations. When constructing the integrated system, we obtain a clear understanding of the integrated information system that we are building. We also need to know how the existing applications map to the newly defined integration architecture and which functionality we will be able to reuse. Then we build the design model, write the implementation code, and perform testing and verification. At the end of the construction we verify whether the developed integration satisfies the requirements.

In transition we deploy the developed integration components into the production environment. Upon deployment there are often additional problems and complications that arise, which we have to solve. The transition usually begins when we have a beta version of the integration components ready. Transition finishes when we are satisfied with the functionality of a certain integration phase. After transition we usually proceed to the next integration phase (from data-level to application interface level for example).

The integration process differs from the usual software development process in that it has to take existing applications into account. Analysis of existing applications has to be made and the integration design discipline has to be adapted. In this article we will focus on both mentioned disciplines:

- Analysis of Existing Applications
- Integration Design

Selection of integration infrastructure has been addressed in [17], the integration assessment in [19]. Other disciplines, such as requirements gathering, problem domain analysis, implementation, testing, and deployment do not differ considerably from general software development disciplines, as described in [18].

3 Analysis of Existing Applications

Before we start analyzing existing applications we have to select the applications to be integrated. This should include all the major primary "backbone" applications. But we should also not forget subsidiary applications, often self-made or locally developed solutions that users use on a daily basis.

In the analysis of existing applications, we identify and specify the functionality of each application that will be included in the integrated information system. We identify the data models, perform the functional analysis, identify the architecture of existing applications, and identify ways to access this functionality.

We also need to identify redundancy and other semantic problems, where the functionality of several applications may be overlapping. Usually in this discipline we will look at the applications in two ways. First, we'll study the data that is stored in applications. Second, we'll identify the functionality that is provided and the ways in which to access it – we will extract the business rules that are embedded in the existing applications. The outcome is the data- and functionality-level analysis models.

We perform the analysis of existing applications in a controlled and disciplined manner and follow the following activities in order to analyze each existing application selected for integration: functional analysis, technical analysis, analysis of functional overlapping, analysis of existing integration. Figure 2 shows the main activities and their refinements.

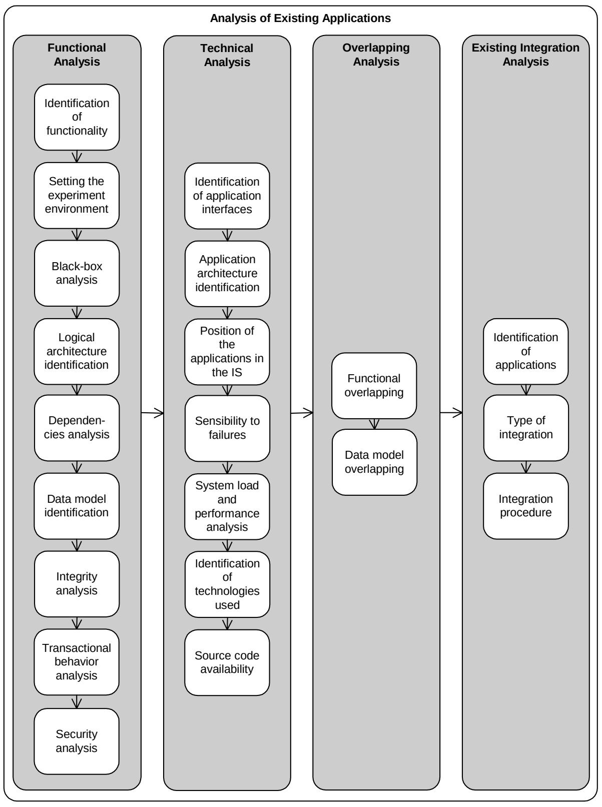
Analysis of Existing Applications
Functional Analysis
Identification of functionality
Setting the experiment environment
Black-box analysis
Logical architecture identification
Dependencies analysis
Data model identification
Integrity analysis
Transactional behavior analysis
Security analysis
Technical Analysis
Identification of application interfaces
Application architecture identification
Position of the applications in the IS
Sensibility to failures
System load and performance analysis
Identification of technologies used
Source code availability
Overlapping Analysis
Functional overlapping
Data model overlapping
Existing Integration Analysis
Identification of applications
Type of integration
Integration procedure

Figure 2: Analysis of existing applications discipline

3.1 Functional Analysis

3.1.1 Identification of Functionality

In order to reuse as much functionality as possible, it is important to identify all the functions the existing applications possess. In addition, we also have to identify how often a function is used. This is important because the existing application could even have some functions that have never been used. There may be no guarantee that these functions actually work correctly. To avoid unpleasant surprises, it is recommended that we consider only the functions that are actually used and which we know work correctly.

The documentation that will be interesting for the identification of functionality includes requirements specifications, analysis and design documentation, testing documentation, and user documentation. For commercial applications, we'll probably have up-to-date documentation, or at least the user documentation (user manual) that will explain how to use the application. For applications developed in-house we probably won't have up-to-date documentation, but we may be able to find the requirements specification. The requirements specification is often the basis for getting a software development project approved. This can be a good start, but we still have to check how each function is implemented. If the application is not too old, we may be in luck and the original developers may still be around. They will have the best understanding of the application and it is well worth talking to them about the functions that their application implements.

If we cannot talk with the original developers we have to talk with system administrators and users. System administrators will have an overview of how often the application has been used and where the problems have been. Users will be familiar with the functions. Although this is not the time to start developing code, we may take this opportunity to check whether the source code actually exists, and if so is it in-synch with the executable versions? Many existing applications do not have adequate documentation. For some, even the source code does not exist. Even if the source code does exist we have to check if it is in-synch with the executable versions.

For outsourced applications we are faced with similar problems as with in-house developments. If the outsourced projects have been managed efficiently then there should be documentation available that will be comprehensive and up-to-date. However, many projects have not been well managed and we will not have the documentation. On the plus side, for almost every outsourced project the requirements specification should exist. It usually forms the basis for the contract and for assessing the value of the software development project. Software development companies are also more aware of the importance of documentation.

However, for an outsourced application it might be even more difficult to get in touch with the original developers. They are probably not employed by the same company. Even if they are, they are not likely to want to talk with us for free. This problem is exacerbated when we find out that the consulting company does not exist anymore and it has delivered the executable application without source code.

3.1.2 Setting Experimental Environment

After we have prepared the list of functions with their frequency of use, we have to check each function to get an idea how it works. We do this in an experimental environment that we have to set up. This will basically be a copy of the production environment, which will enable us to experiment with existing applications without disturbing their everyday operation. Setting the experiment environment is important not only for the analysis phase, but is very useful later when we apply modifications to the existing applications. Without an experimental environment it would be absolutely impossible to safely test and validate the integration solutions.

Setting it up can vary in complexity. It is easy in cases where we have the necessary hardware, and where we can simply copy the applications, with or without the persistent data. The more complicated the application architecture, the more work we have to set up the environment. Becoming comfortable in the environment of the existing applications is crucial to achieving integration.

This will be the most difficult for legacy applications. For them, there will be the problem of obtaining the necessary hardware, and we probably won't be familiar with the environment and the tools, which may present the biggest obstacle. A big problem can be setting up experimental databases. Again it depends on the architecture of the application: if it uses some standard way to access the database it will be easier.

Only with commercial applications we expect to have some form of installation procedure. However, we have to be sure that the actual product is identical to the application that is used in production. Otherwise it is a better idea to use the production version.

For applications where performance workload is not limited, we are able to use the same hardware for the production and experimental configurations. If we make this decision, we have to be very careful not to interfere with the production data. This approach will not be applicable if the application has a high workload and/or is mission critical. In this case we have to set up a fully isolated experimental environment.

If we are unable to set up an experimental environment for an application we want to integrate, we have to be very careful with the tests that we do. We have to consider what time to perform the tests, for example, when the application is not in use (during nights, weekends, or holidays). This will influence our flexibility considerably.

3.1.3 Black-box Analysis

We then have to check each function that we have listed in the functional specification. Note that we're not only talking about the functions accessible from the user interface, we have to include all functions, even those that the application provides through APIs.

We call this activity black-box analysis because we don't care about how the function is performed by the existing application. We are interested solely in the output that we get and what input parameters we have to provide to get the desired output.

When specifying the input and output behavior we should pay particular attention to the boundary conditions. This means we should consider the allowed intervals for input parameters. We should specify this in the form of preconditions for the input parameters. This will become important later when we reuse the functionality.

To describe the functions of existing application we can use a textual form, where we produce a table and description of the functions. The proposed table should include the following columns:

- Function – name of the function that the application provides.
- Description – description of the functionality.
- Access via user interface or via API – we should identify how we can access the functionality.
- Frequency of use – we should identify if the function is used at all and, if so, how frequently it is used. If possible we should use an objective metric, for example number of times per week.
- Required inputs – we should clearly identify the input parameters and their allowed ranges.
- Outputs – we should identify the outputs that we get.

3.1.4 Logical Architecture Identification and Dependencies Analysis

After we have identified the functionality of the existing applications, we have to recognize its internal structure. Here we first have to identify if the application is monolithic, client/server, or multi-tier. Then we try to categorize how it is constituted – if there are several modules or components, where the business logic is, etc.

After we have identified the logical architecture we have to classify the dependencies between the applications. Here we should identify all the dependencies. Two applications can have logical dependencies that can be implemented either automatically or manually.

If implemented automatically then there is a sort of interoperability between the applications – these applications share data or functionality. Often, particularly with legacy systems, such connections are implemented through data exchange, very often via shared files or tables. This will be important later when we come to identify the existing integration between applications. Then we will consider how the integration is implemented from a technical perspective.

More frequently, we will see dependencies that are carried out manually. This means that the users will have to re-enter the same data, leading to possible inconsistencies. An application can provide a summary of some data it processes that the users then enter into some other application. There is obviously a dependency between them that we should identify and show on a diagram. If possible, we can also document these dependencies. This information will be useful later in the analysis.

3.1.5 Data Model Identification

Another very important activity in functional analysis is the identification of the data models used by each application. This is important because we have to understand how data is stored. We have to analyze the persistence storage of each application. We will be faced with one or more of the following types of databases: relational, object-oriented, universal, multidimensional, hierarchical and network, other formats, such as flat files.

We have to construct the database model for each existing application. This will be the basis for data-level integration. Often it is possible to generate database models automatically with the tools provided by the database. The majority of relational databases, for example, have tools to generate entity-relational (ER) schemas out of existing databases. This is usually better that depending on possibly out-of date documentation.

3.1.6 Integrity Analysis

Here we identify how the integrity of databases is achieved and which party is responsible for it. Most likely each application will be responsible for assuring the integrity of their own databases. In this activity we should identify the integrity rules for each database that the system uses. Identifying the integrity rules will be particularly important for data-level integration when we exchange data between applications based on direct database transfers. Since we will most likely omit the business rules at this stage, we have to be aware what the integrity rules are.

The integrity rules are sometimes described in the documentation. Sometimes they are incorporated within the database, if the database allows this. More often these rules are coded within existing applications. Database administrators can be very helpful with the identification of integrity rules.

The problem with the identification of these rules is that it is very difficult to be sure that we have identified all of them. Not identifying them on the other hand can lead to breaking the integrity of databases. Identifying this problem is a difficult task, and tracking down failures to database integrity problems is very time consuming.

3.1.7 Transactional Behavior Analysis

Transactions play an important role in all non-trivial applications. Their management is known as transactional processing. Transaction monitors can be a DBMS or some dedicated middleware. Transactions can work with a single resource – these are the simplest and most commonly used. However, in large systems the transaction might need to be invoked over several systems. This is when distributed transactions come into play. A distributed transaction spans more than one resource. Their context can be propagated or shared by more than one component; they require the cooperation of several different transaction monitors.

Our goal will be to identify the transactional model (flat, nested, chained or saga) and become familiar with how it works together with the existing application. We have to familiarize ourselves with transactional properties of the existing application, identify how the existing application uses transactions, and how critical the failures are.

3.1.8 Security Analysis

In security analysis we have to examine the way that security is utilized by the existing applications. Generally we need to answer: Does the application implement security? If yes, how is the security implemented? If no, should we add security now? There are four important security mechanisms found in existing applications. Authentication is the process of verifying that a client is who they claim to be. It can be performed on the client before it interacts with the server. It can also be performed on the server.

Authorization checks whether the client application is allowed to perform a certain operation. Authorization can be defined programmatically or declaratively, depending on the implementation. Typically it is defined in terms of security roles and Access Control Lists (ACLs). Extracting info on how authorization is performed from existing applications can be complicated because the logic may be in the application code.

Communication channel security – newer applications will typically use Secure Socket Layer (SSL) and Transport Layer Security (TLS), but this can differ significantly with older legacy systems.

Auditing let us see an exact history of operations performed on the system and is useful for analysis of past events.

The fact is that a lot of existing applications do not have much security implemented. Therefore attention will have to be paid to how to introduce security to existing applications.

3.2 Technical Analysis

3.2.1 Identification of Interfaces

In this activity we focus on the application interfaces. Our goal will be to specify the interfaces that an existing application provides to other applications. First of all, we have to identify how many interfaces there are and which operations they provide. Then we have to identify which technology is used to access them.

To identify the number of application interfaces we will have to go through the documentation, talk with the developers, and even analyze the source code. We might also consider using tools for analyzing existing applications. Such tools sometimes can identify application interfaces even if no source code is available. We should mention that we could consider every form of communication between two applications as an application interface. For now it's not important if those interfaces are implemented in a proprietary technology, if they are procedural or functional, even on protocol level.

We specify the interfaces on the UML component diagrams using the interface stereotype. We also identify the operations of each interface and show their signatures. This means that we have to identify the names and the syntax of operations, the necessary parameters and the return value.

Sometimes we have a situation in which the applications are tightly coupled, so there will have to be some preconditions fulfilled before an operation can be called or invoked. We need to identify these preconditions (and maybe post-conditions). We also try to identify if there are some restrictions in the order in which the operations have to be invoked. Another important thing is to recognize the way that the application signals errors or other exceptional conditions.

Identifying the interfaces is very important, particularly for application-to-application integration, but sometimes also for data-level integration. Accessing data through operations is better than going directly to the database because we avoid circumventing the business logic. This makes it easier for us to maintain database integrity.

3.2.2 Architecture Identification

Having identified the logical architecture and the interfaces, we should now consider the physical architecture. We need to become familiar with the environment in which the production application is deployed, so we identify the computers on which the application parts are deployed and the type of connection between them. This step should be done for each application separately, although applications will frequently share resources.

To represent the architecture we can use UML deployment diagrams. They show the runtime configuration of hardware devices and the software components that execute on them. Nodes contain component instances, which show that the instances execute on a certain node. Typically there will be several component instances on a single node; however this depends on the granularity of the application. Monolithic and client/server applications will be typically represented by a few components only.

It is also very useful to show the dependencies between the component instances using a dependency relationship. If the components provide interfaces that their communication relies on, then we should show the dependencies using the interfaces that we have already identified. For example, an existing application can provide a custom API for communication with clients, and clients can use a remote procedure call or message-oriented middleware to call the procedures and functions in the API. This can be seen as an interface although it is not an interface in the sense of component/OO-based development. If there are no interfaces that we can identify, then we should just show the dependencies between the components. Sometimes we can specify the communication protocol for each dependency too.

After we have identified the architecture of each application separately we should build the diagram of the whole existing information system. This basically means that we gather together the deployment diagrams that we drew in the previous step. We also need to identify which resources the application share and denote the dependencies (already identified previously) on this diagram.

We should extend this diagram with the other existing applications that are present in the current information system, but have not been selected for integration. We should mark them clearly with the <<external>> stereotype, and note whether there are some dependencies between the external applications (those not selected for integration) and the applications that we are integrating.

Sensibility to failures analysis is the next step in the technical analysis of existing applications. Here we have to identify how critical each application is for the company. We have to see if the company has alternative scenarios regarding what to do if an application fails. If it does not (and most do not have such scenarios), we must develop them. Note that when altering an existing application we will considerably increase the risk of failing, so we have to take every measure possible to minimize the risk. This includes efficient backup systems, which include application data as well as the executable application files.

3.2.3 Performance Analysis

Here we should clarify what the performance considerations of applications are. In the requirements gathering phase we should have already identified the performance expectations for the integrated system. Here we have to see how the existing applications perform. When integrating applications, one of the goals is to provide instant access to information. The technical implications are that after integration there will be a larger number of clients that will simultaneously use the application. Sometimes, for example when making applications accessible online, this number can be considerably higher.

It would be wrong not to consider the performance limitations now. We will look at the system from two perspectives: the client load, that is, the number of concurrent clients, and the data load, that is, the quantity of persistent data.

To identify the client load we should look at the predicted average and maximum number of concurrent clients; the response time by average and maximum number of clients; the highest acceptable response time; how much we increase the number of clients to fulfill the response time limit in the current configuration; the possibilities there are to increase scalability (hardware and software solutions).

If we identify that the application currently offers acceptable response times (and it should, because this is a production application, although in real-world it often does not), we will try to identify how much potential there exists in the application for raising the number of simultaneous clients. From this, we will try to infer the highest possible number of concurrent clients that the existing system can support in its current configuration.

To identify the data load we will first assess the current persistent data size. Then we try to identify if the integration will increase the data size. The reasons can be different. For example, it is possible that pre-integration the data between applications is transferred only once per month and only summary values are recorded. Upon integration we may require this transfer several times per day or even instantly. This will also mean that the integrated system will record each transaction separately, thus increasing the persistent data dramatically.

3.2.4 Identification of Technologies

In this step we have to familiarize ourselves with the technology used in each existing application. If this is a commercial application we have to check the exact version that is being used. If it is a custom-developed application we have the following points to check: programming language, compiler, IDE, linker, operating system version, DBMS versions, middleware, and all other related software. We also have to look if those versions of software still exist, and if not, how we can obtain them. This will be important for making decisions on rebuilding the system using the source code.

After we have defined the technology we have to see if the source code is available for the existing application. There are a large number of systems (particularly legacy) where source code is not available. Source code will also probably not be available for commercial applications.

For custom-built applications, we will most likely have access to the source code, unless they are old or the source code has been lost, be it accidentally or intentionally. But even if we find the source code we have to check that we have all the necessary tools to rebuild the application and the source code version corresponds to the actual version used in production. Often it happens that a single missing library or configuration file prevents us from rebuilding the application.

To check whether the production version is identical to the source code we can use a simple procedure. We build the application from the source code and compare it to the production version using a file compare utility. We have to be sure that we compare the executable files only, without any data. If this simple procedure does not work then we will have to compare applications, which can be very difficult for small changes.

3.3 Overlapping Analysis

After we have analyzed the existing applications from functional and technical perspectives, we are familiar enough with them to perform an overlapping analysis. The objective here is to identify which parts of the applications overlap – which functionality and data is redundant. We also select which application is responsible for which overlapping functionality. Overlapping analysis consists of two steps: functional overlapping, and data model overlapping.

3.3.1 Functional Overlapping

As existing applications are not usually integrated, an application can often contain certain functionality that has already been implemented by some other application. This is essentially due to a lack of architecting.

So, we are often faced with two or more applications that implement the same functionality. Often one application implements it in the detail, while another implements only the parts that they need. Typically these applications will introduce a slightly modified view of the functionality, which will complicate the situation even more.

We would like to identify which functionality is overlapping in the applications that we have selected for integration. Now we identify which functions of which existing application we will use later, when we reuse some existing functionality for the integrated information system.

To identify the overlapping functions it is a good start to have a look at the dependency analysis that we performed as part of the functional analysis. We should look at the dependencies; particularly those that are implemented manually are suspicious. Implementing a dependency manually means that the user has to re-enter some data that has been processed from one application into another. This may mean that the applications had to overlap a part of their functionality. We also have to check the dependencies that are implemented automatically. If there is only data exchange between applications it can still mean that the functionality is overlapping.

To describe the functionalities that are overlapping we first have to identify the function, then all the applications where the function is implemented and finally select the application that will be responsible for that function (the application that we will use when reusing this function for integration).

3.3.2 Data Model Overlapping

After we have analyzed the functions, we also have to identify the data that might be overlapping in the databases of different applications. Dependency and functional overlapping analysis can be useful here. Functional overlapping almost always means that there is data overlapping under it. But note that there might be data overlapping somewhere else, too.

To identify it, we should again focus on identified dependencies between applications and evaluate first those implemented manually and then those implemented automatically. For data model overlapping analysis, it is very helpful if we have the schemas of all the databases. Then we can identify the data that is overlapping. Similarly, as in functional overlapping, we should select the databases that will be responsible for certain data. These databases will then be used in the integrated information system.

If we are lucky we will only have to deal with one database model, probably relational. Then we have to identify the entities that are overlapping. If we build the data dictionary is very useful to supplement the information that each entity name represents. This point it is also a good opportunity to resolve the name conflicts and to explain the cryptic names for entities and attributes. The most difficult task will however be to resolve semantic issues.

3.4 Existing Integration Analysis

The last activity is to identify any existing integration solutions. It is very likely that we will be faced with some form of already implemented integration. The most common ways are data exchange using shared databases or flat files, or the use of message-oriented middleware to enable point-to-point communication between applications. We have to be aware of existing solutions when planning our integration, although it is often simpler if we don't have any integration at all and can start from scratch.

First we identify all the applications that each application is integrated with. As we have already done the dependency analysis this will not be very difficult. We pay attention to all automatic dependencies, and focus on some specific details that we need to identify: type of integration, exact procedure of how integration is implemented and performed.

The type of integration identification is the second step. We will identify what integration level the existing integrated applications use.

4 Integration Design

In this discipline we focus on the global architectural design model, where we represent the integrated information system as a set of components (identified in the problem domain analysis discipline) that have well defined interfaces through which they communicate. Instead of focusing on how to implement each component from scratch, we focus on how to reuse existing applications to provide implementations for the components.

We approach architectural design from a high-level perspective. Due to the size and complexity of the problem domain, it is practically impossible to design the integration architecture down to the finest detail. This would also be unreasonable because a lot of functionality is implemented by existing applications. Accordingly, we approach the architectural design in a more high-level way, where we define the global architecture in the sense of components and their interfaces. This is somewhat analogous to the planning of a city's architecture compared to designing a house.

Several key activities of this discipline characterize the architectural integration design process. Firstly, we cope with the global situation, and then we focus on information system-specific-functions. Here we start to solve the use cases that influence the architectural decisions and, as a result, we produce a set of subsystems. Each of the subsystems realizes a use case. After iterating though the subsystems we start building the global architecture step-by-step and finally define a stable architecture.

The main activities of the integration design discipline can be organized into three groups as shown in Figure 3.

The integration design discipline is a highly important discipline. Getting the integration design wrong will result in the failure of the whole integration project. Of course the quality of the results in the design discipline is dependent on the quality of the inputs from previous disciplines. Still we should be aware of the importance of this discipline. The risk of mistakes can be greatly reduced with iterative and incremental development.

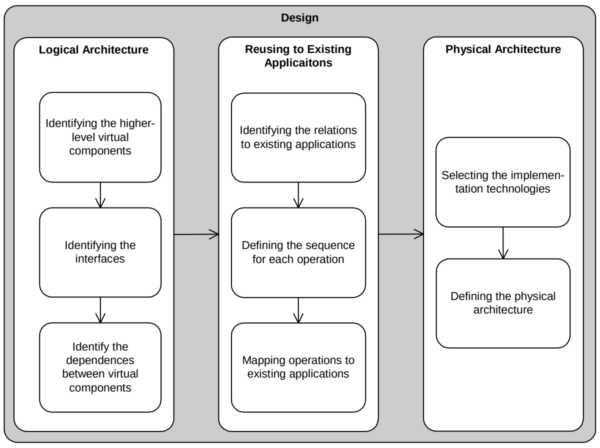
Design
Logical Architecture
Identifying the higher-level virtual components
Identifying the interfaces
Identify the dependences between virtual components
Reusing to Existing Applicaitons
Identifying the relations to existing applications
Defining the sequence for each operation
Mapping operations to existing applications
Physical Architecture
Selecting the implementation technologies
Defining the physical architecture

Figure 3: Integration design discipline

4.1 Logical Architecture

Identifying the higher-level virtual components [20] is the first activity in the integration design discipline. We need to identify the higher-level virtual components that constitute the integrated system. The problem is that although this task sounds easy, in reality it is not.

Selecting the correct higher-level virtual components will have a long-lasting influence on the information system as a whole. The selection also determines how suitable the integration architecture is to re-engineering existing applications and replacing them with newly developed solutions.

To identify the high-level virtual components, we focus on the analysis model class diagram. The analysis-level entity and control components that we identified will map to virtual components on the business logic tier, so we will focus on them. The analysis-level boundary components represent user interface constructs. These will be realized in the client and web component tiers.

To identify the virtual components we go through the control and entity components from the problem domain analysis discipline. We try to group them into virtual components based on their functionality. Components encapsulate their internal implementation and represent their functionality through the interface. To identify the higher-level virtual components we can follow these guidelines:

- Start with the analysis class diagram.
- Gather the analysis components that are logically connected because they implement a part of a larger functionality.
- Try to make the virtual components as independent of other components as possible.
- Often we will have to add other specific components that will implement non-functional requirements, for example, or model some implementation-related concepts.

After we have identified the higher-level virtual components, we define the interfaces through which we access the functionality of these components. We should ensure that the interfaces are high-level and that they focus on business processes and not on implementation details. The interfaces act as the contracts between the components. The interfaces represent a part of the integration architecture that we should not change – each change will influence all dependent components.

Keep in mind, however, that we can still add operations to existing interfaces without creating problems on related components. Therefore we will often introduce modified methods as new methods with a slightly different signature. This protects us from having to change all related components. However, doing this too many times will make the interfaces very hard to use because we will have to cope with the redundancy of methods – we will not know exactly which to use and when. So we have to be very cautious with the interfaces that we define.

Identifying the dependencies is important because they show how the changes to one part of the system will influence other parts. Dependencies between parts of the system can be direct, in which case a change in one part will require a modification to another part. For example, if part A is directly dependent on part B, this means that if we change something in B we also need to update A.

Dependencies can also go through interfaces, which will decouple the direct connection between the two parts of a system. This will obviously be the preferred way and we will model the integration architecture through interfaces, as we have already stressed several times over. Making components dependent only on component interfaces simplifies their management considerably. As long as we do not modify the interfaces we can change the implementation of the component.

Still, we have to be aware which dependencies exist between virtual components, so we will identify them and show them on the diagram. This enables us to efficiently track and measure the complexity. As we apply changes to the architecture, we should also update these diagrams, otherwise they are effectively useless.

The degree of coupling between components can be used to identify and describe the dependencies. Weak coupling shows that the groups are relatively independent, and fewer dependencies between components show that we have gathered the classes correspondingly and that the system will be relatively easy to understand, maintain, and extend.

Strong coupling, on the other hand, indicates that there are many dependencies between components. This suggests that changes to one part of the system (to an interface, for example) will require modifications in many other parts. It also makes the structure of the system less easy to understand. Sometimes strong coupling is a consequence of incorrectly gathered classes and poorly identified components, and in such cases, it might be a good idea to rethink the architecture. Indeed, such re-evaluations can be a normal part of the whole process.

4.2 Reusing Existing Applications

Identifying the relations to existing applications is the first step in this activity. It is recommended to show the relations for each component, because this will make it easier to follow later steps. This stage is dependent on the existing applications that we have. To be able to identify the relations to existing applications we have to be familiar with their functionality, and to achieve this we have to do the analysis of existing applications.

When we have identified the existing applications that the higher-level virtual component has to interact with, we identify the exact sequence of operations that the higher-level virtual component has to invoke in order to get the desired result. To identify the operations and the sequence that needs to be invoked we study the interfaces of existing applications lower-level virtual components and map the desired functionality in the best possible way.

In real-world examples we will frequently be overwhelmed with the complexity of the interfaces that existing applications provide. We will often also be confused about which operations to actually use, because often there will be more than one way to achieve the same result. To model the sequence of operations that have to be invoked we can use UML sequence diagrams. It is very important that we model all possible sequences of operations, including the normal flow of events and any alternative flows in which something could go wrong. In this way we can define how to handle all exceptional situations, how and to whom we should propagate the exceptions, and we will ultimately make our components highly robust.

The sequence of operations sometimes is not enough and the component has to do some calculation, and perform other operations to get the desired result. As such, in this step we must identify what exactly has to be done. The goal is to identify the interaction with the existing application to such a level that we will be able to write code directly from the specification.

It will vary from operation to operation how complex a mapping we will have to use. With a highly complex mapping we might consider representing the whole procedure with an activity diagram too; sometimes we could even use "pseudo code". We have to map each operation of the newly defined higher-level virtual component to lower-level virtual components that represent existing applications. Sometimes we will not be able to find the corresponding methods in the existing applications. This means that the functionality we require is not supported by existing applications, in which case we have to implement it from scratch. Or we might be able to reuse only a part of the whole functionality. Following the proposed integration process we will be able to add the missing functionality in a relatively painless manner.

4.3 Physical Architecture

In this activity we have to select the implementation technologies and physical architecture. The selection of implementation technologies will depend of the used software platform. We have to take into account the requirements regarding performance and reliability. This will then influence the deployment scenarios that we select.

To achieve acceptable performance we consider locating tightly-coupled components inside a single container and use local access to components to optimize the method invocation performance [21]. To achieve higher reliability we might consider clustering or replication.

To identify the most suitable physical architecture we select a few different candidate architectures first. Then we build prototypes that help us to validate these candidate architectures by the criteria that we have to meet. Only then will we select the final appropriate architecture and do the implementation.

5 Conclusion

In this article we have presented the process model proposal for information systems integration that specifies a disciplined approach to top-down integration. The integration process introduces sound practices, like iterative and incremental development, prototyping and reuse. It specifies the phases, disciplines, and activities. The four integration phases are: data-level, application interface level, business-method-level, and presentation-level phase.

For each phase the integration process defines several disciplines that have to be performed in order to obtain results. Some of these disciplines are equal for all phases, some depend on the phases. We have focused on the technical disciplines only.

Analysis of existing applications and integration design are highly important disciplines for integration projects. We have to get a clear understanding of the existing situation in order to be able to later map the functionality to the newly integrated system. We also need to adapt the design phase to involve existing applications. This is why in this article we have focused on those two disciplines and presented detailed activities which should be carried out as a part of each discipline.

One of the important features of the presented integration process model is its ability to be adapted to specific need of each company, which will be addressed in our future work.

References

[1] J. Lee, K. Siau, S. Hong (2003) Enterprise integration with ERP and EAI, *Communications of the ACM,* ACM, Vol. 46, Iss. 2, pp. 54 – 60.

[2] M. Stonebraker (2002) Too much middleware, *ACM SIGMOD Record,* ACM, Vol. 31, Iss.1, pp. 97 – 106.

[3] J. Sutherland, W. J. van den Heuvel (2002) Enterprise application integration and complex adaptive systems, *Communications of the ACM,* ACM, Vol. 45, Iss. 10, pp. 59 – 64.

[4] Z. Wu, S. Deng, Y. Li (2004) Introducing EAI and Service Components into Process Management, *Proceedings of the Services Computing,* IEEE, Shanghai, pp. 271 – 276.

[5] A. Eyal, T. Milo (2001) Integrating and customizing heterogeneous e-commerce applications, *The VLDB Journal,* Springer, Vol. 10, Iss. 1, pp. 16 – 38.

[6] S. Baker (2002) The three steps to web service integration, *IONA,* www.iona.com.

[7] F. Losavio, D. Ortega, M. Pérez (2002) Modeling EAI, *XII Int. Conference of the Chilean Computer Science Society,* IEEE, Chile, pp. 195 – 204.

[8] I. Gorton, A. Liu (2004) Architectures and Technologies for Enterprise Application Integration, *26th Int. Conference on Software Engineering,* IEEE, Edinburgh, pp. 726 – 727.

[9] P. Maheshwari (2003) Enterprise Application Integration using a Component-based Architecture*, 27th Annual Int. Computer Software and Applications Conference,* IEEE, Dallas, pp. 557 – 560.

[10] V. S. Pendyala, S.Y. Shim, J. Z. Gao (2003) An XML Based Framework for Enterprise Application Integration, *Int. Conference on E-Commerce,* IEEE, California, pp. 128 – 133.

[11] D. Gawlick (2001) Infrastructure for Web-based Application Integration, *17th Int. Conference on Data Engineering,* IEEE, Heidelberg, pp. 473 – 477.

[12] B. Hunter, M. Fowler, G. Hohpe (2002) *Agile EAI Methods: Minimizing Risk, Maximizing ROI,* ThoughtWorks Inc.

[13] S. Chatterhee (2004) *Managing EAI Projects in Agile way,* Cap Gemini Ernst & Young Consulting.

[14] M. Fowler, G. Hohpe (2002) *Agile EAI,* ThoughtWorks Inc.

[15] G. Hohpe, W. Istvanick (2002) *Test-Driven Development in Enterprise Integration Projects,* ThoughWorks Inc.

[16] D. S. Linthicum (1999) *Enterprise Application Integration,* Addison Wesley.

[17] M. B. Juric et al. (2001) *Professional J2EE EAI,* Wrox Press Ltd.

[18] M. Silic et al. (2000) *EMRIS - Enotna metodologija razvoja informacijskih sistemov, Zv. 4, Objektni razvoj,* Center Vlade RS za informatiko.

[19] M. Pusnik, B. Sumak, M. B. Juric, M. Hericko (2004) Ocenjevanje pripravljenosti podjetij na proces integracije s pomočjo indeksa integrabilnosti, *7th Internation Multiconference Information Society IS 2004,* Inštitut Jožef Stefan, Ljubljana, pp. 53 – 56.

[20] M. B. Juric et al. (2003) Application integration patterns, *Technology supporting business solutions, Advances in computation: theory and practice,* Nova Science Publishers, New York, pp. 115-138.

[21] M. B. Juric et al. (2002) *J2EE Design Patterns Applied,* Wrox Press Ltd.