UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science

Siim Karus

Forward Compatible Design of Web Services Presentation Layer

Master's Thesis

(20 ap)

Supervisor: Professor Jüri Kiho

Author: ……………………………………..	"…..“ May	2007
Supervisor: …………………………………	"…..“ May	2007

Tartu 2007

Table of Contents

1 Introduction

1.1 Aim of the Study

While web services become more and more personalized and the information has to be presented for various channels, the load on services presentation layer increases. In addition, any new features or changes in present features may require changes in all presentations and presentation layer components. These changes can take up majority of development time.

It is very common to use XSL transformations [1] to create the actual output. This paper discusses different solutions used for lowering the cost of making changes in presentation layer and gives a framework on how present tools can be used in a cost-effective way. Additionally, an example of real solution using XSLT is shown.

Web services are constantly evolving. Unlike other software projects, web services development never stops because new features are being requested by the users or demanded by law. When services grow older and more complex, responding to bug reports and fixing bugs gets more expensive. Users might also decide to require redesign of present features to allow use of newer, modern user interface features and design concepts.

With the intention of avoiding high costs of presentation layer, service owners often limit the number of output channels or different presentations of service data. In some cases, this can be very successful and effective solution; however, it cannot be applied always. For example, public services like MSN Spaces, Blogger or VabaVaraVeeb have found customizability of user pages a very popular and important feature. Even though generally only CSS and DHTML based customizations are used, XSLT used by some services allows extremely extensive customization. Thanks to XSLT and xslt-req[2], new presentations could even be created by the users with almost no review required by the service provider. However, most of the users are not willing or able to support the designs or outputs they have created and the providers cannot support them because of the costs. Therefore, the presentation layer components, which take business input and generate output, must be created in a flexible, forward compatible way.

At present, forward compatible design is becoming an important topic. However, there exists no general solution or framework, telling how forward compatible applications and components should be made. The aim of this thesis is to propose a solution for creating forward compatible presentation layer components and to give an example of this solution put into use.

1.2 Structure of the Thesis

The first chapters (2-3) of the thesis identify the main problems of forward compatibility and the costs of making changes in services. The next chapters (4-5) analyze present solutions and frameworks used to solve any of the problems presented in the first chapters. Next, based on the present solutions, requirements for the general solution to solve problems stated are formed (chapter 6). Solutions to solve remaining issues are proposed. Based on these propositions, general method for creating forward compatible presentation layer components is composed (chapter 7). Finally, based on the general method, a solution for building XSLT based forward compatible presentation layer components is shown and analysed (chapter 8).

The language of the thesis is English, because Estonian terminology is not present at the time the thesis is written. Nevertheless, translations of the core terminology are proposed in the appendix III.

2 Background

2.1 Forward Compatible Design

Most web services today are backward compatible. They show fine on almost all older browsers and clients. Any newer components are introduced carefully by supplying a fallback to traditional components. Backward compatibility is something we have grown to expect from any service and non-compatible services will suffer from low interest. Most services are designed with backward compatibility in mind.

However, forward compatibility is often overlooked. It has become the difference between successful services and fading services. Forward compatibility assures easy extending of present services, viewability on new devices, connectability with other services and long life of services. Forward compatibility of an application is its preparedness to changing environment and adoption of technologies.

Sometimes forward compatibility seems to come free. Popular technologies and ideologies like object oriented design and modular, extensible design already do supply some forward compatibility. Often, these are sufficient, but in some cases, more effort needs to be put into forward compatibility. According to Chris Armbruster[3], three design principles to assure forward compatible design are extensibility, abstraction and componentization. In his internet white paper on forward compatible design, he also provided five fundamental questions to be asked when designing new service:

1) What happens to my application if the business rules change?
2) What happens if we enter a different market place?
3) How much additional work will be required to incorporate new technologies like speech recognition, natural language query and hand writing recognition?
4) What about Internet devices that do not look like computers such as Web TV, PDA’s and cellular telephones?
5) What if the technology for storing data changes?

Even though the paper comes from the end of 1990s and meanwhile new technologies have appeared, these questions have remained relevant. All these questions help to understand the relevance of forward compatibility.

2.2 Extensibility

Extensibility of an application is its capability to be extended. Extensibility is important factor of compatibility as it allows applications to support older technologies (backward compatibility) and newer, not yet available technologies (forward compatibility).

It is easier to create backward compatible applications, as information about their requirements is already known. Therefore, backward compatible applications can be designed non-extensible. According to Armbruster, in order to support future technologies, applications must be able to negotiate usage-level details and degrade gracefully as new technologies come to use.

2.3 Abstraction

Abstraction is withdrawal or separation of components. An abstracted architecture is an architecture where all sub-components are isolated from each other and from the whole. Abstracted architecture is used to allow independent development of different components of an application. It is especially important when making changes in applications as required changes will have to be made in application layers affected with the required change.

n-tier Applications

An *n*-tier application is a term used to describe modern client-server applications, which do no longer have distinct server or client components. From functional viewpoint, applications have three layers: presentation layer, business logic layer and data services layer. *n*-tier applications abstract or isolate components in all these layers making these three layers independent from each other. The isolation results in more independent virtual layers. This should allow making business logic changes by just modifying business logic layer and not modifying presentation or data services layers. This kind of architecture allows using and later adding of multiple different user interfaces, logic modules and database services. Relation between n-tier and functional viewpoint is illustrated on figure 1.

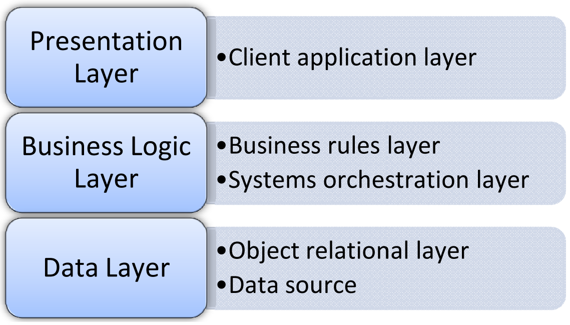

Figure 1. Classical functional viewpoint relation to n-tier viewpoint

2.4 Componentization

Componentization is separation of an application into separate executable components. Componentization describes components in a way that allows independent upgrading or replacing of different components. This makes evolution on component-by-component basis possible.

Armbruster declared five requirements for components of modern applications. These requirements are:

- Language independence
- Shippable in binary form
- Upgradeable without breaking old clients
- Transparent location and relocation on network
- Dynamic linking

2.5 Future Proof Design

A solutions design is future proof if the solution can be used with future technologies without having to make any changes to the solution itself.

3 Types of Changes in Software

As software matures, changes need to be made. Mikael Svahnberg and Jan Bosch [4] categorized the ways of software evolution as following:

- improvement of functionality;
- changed component to support product change;
- new framework implementation related to infrastructure change;
- changed framework implementation;
- new component to support product change;
- replaced component to support product change;
- split of software product line;
- derivation of product line architecture;
- split of component;
- new relation between components;
- changed relation between components;
- decreased functionality in framework implementation; and
- solving in external component.

Even though, the improvement of functionality, changed component to support product change, and new framework implementation related to infrastructure change are most frequent, they are not as expensive as changes in architecture. The study showed that the impact of these three most frequent categories of changes can be estimated fairly accurately.

The study also mapped new requirements to the evolution categories. The studied requirement categories were:

- new product family;
- new product;
- improvement of functionality;
- extend standard support;
- new version of infrastructure; and
- improved quality attribute.

The relations of requirements categories and evolution categories can be seen in table 1.

4 Types of Components

In order to be forward compatible, components need to know about possible changes in their communication protocols. In fact, they need to know, which methods are available to them. The need of this kind of information is especially apparent when designing the presentation layer. For example, the presentation layer needs to know how the user can interact with the business logic components.

Solutions built using ASP.NET Web Forms are encouraged to solve this issue by using Web Services described using WSDL (Web Services Description Language) [5] to communicate with the business logic layer. The service description gives the presentation layer the information needed to decide on the features and protocol that can be used for communicating.

However, presentation layer components do communicate not only with business layer. They also communicate and relate to each other. This is where the structure of presentation layer becomes important.

From the behavioural viewpoint we can differentiate between three different types of presentation layer components:

- Controls used to display or edit some specific piece of information;
- Containers used to group together different widgets or containers to present related sets of information;
- Services used to provide means for generating specific type of renderings (e.g. rendering documents to PDF for printing or rendering XML format or EDI format messages to be used in business-to-business scenarios).

These component types can be considered as different levels of presentation layer components as they usually form a hierarchy where services are at the root and widgets are leaves.

5 Present Solutions

5.1 Controls level solutions

Common Controls, ActiveX

A common way of separating presentation layer from business logic is making use of using Common Controls. Common Controls are presentation level components used to perform common input and output [6]. Common Controls save developers form recreating dialogs, fields or other common presentation layer components [7]. On the other hand, they allow developer to do some fine-tuning on them. Due to their simple interfaces, they are easy to extend and can even be combined to create new components. These properties make Common Controls forward compatible, which has contributed to their longlivety and wide range of application.

However, in order to be meaningful, Common Controls need another presentation layer control, container or window to layout them and process their notifications.

Widgets

A widget is a combination of a graphic symbol and some program code to perform a specific function [8]. Even though generally widgets are not required to be extensible and make use of componentization, desktop environments like KDE have begun to include widgets for common tasks in a similar way as Windows uses Common Controls. These widgets have the same benefits as Common Controls, but are by definition limited to graphical interfaces for displaying them.

JavaBeans

JavaBeans is the component architecture for the Java 2 Platform, Standard Edition[9]. Java 2 Platform, Enterprise Edition uses Enterprise JavaBeans instead. Even though JavaBeans is a more general solution, it is most commonly used when building applications or services presentation or persistence layers. JavaBeans are used to encapsulate objects into a single object called the bean. Beans have to obey conventions that allow beans to be manipulated visually in a builder tool.

JavaBeans conventions make them extensible and allow the use of abstraction. Java architecture allows use of componentization at class, package and library level. However, similarly to Common Controls, JavaBeans rely on classes and components that are not beans.

5.2 Container Level Solutions

Templates and Skins

Many content management systems use templates to describe the structure of user interface. Templates define the layout and positioning of control level objects.

Templates for graphical user interfaces are commonly called skins as they change the visual appearance of an application.

Even though skinning is popular, it does not enforce forward compatibility. Most often skins do not support extending and are meaningful only to certain versions of application. On the other hand, there are also templating tools designed for extensibility. One of these is Extensible Stylesheet Language Transformations (XSLT).

Templates can be abstracted because templates can internally make use of other, independent templates to render details of user interface. Also, similar aspects can be rendered by the same template. However, templating tools do not enforce this property and some templaters even exclude it.

Templates can be componentizised as they can be developed as separate components and in separate components. Templating tools do not enforce this property and there are even templating solutions that require templates to have other supporting code (might be even business logic code) written into them.

5.3 Service Level Solutions

Service level solutions are often integrated with business logic. They act as the glue between business logic and presentation layers. The service layer is only needed if the system has multiple external interfaces. In case of only one external interface, one container can be used as the root container for rendering the interface.

The glue between these two layers can be a specialized framework, but might also be something as simple as just using different containers as root containers for different external interfaces.

JavaServer Pages

JavaServer Pages (JSP) [10] is an extension of Java Servlet technology. It is mostly used to bind Enterprise JavaBeans code with HTML code, however, the technology itself supports binding any Java code with any XML-like presentation code. JSP is not a pure presentation layer solution since it also describes the logic that generates the contents of the page. Therefore, JSP can be seen as templating solution that requires at least some business logic code to be written into templates to communicate with business objects. Therefore it is prone to breaking if business logic changes.

FreeMarker

Limitations of JSP have brought us other solutions for gluing EJB with presentation markup. One of these solutions is template-based FreeMarker [11]. Being completely templates-based, it has the benefits of using templates as described before. In conjunction with EJB, FreeMaker can be used to create true forward compatible presentation layers for Java based web services. However, it is important to note that templates do not enforce forward compatibility.

Active Server Pages

Active Server Pages (ASP) [12] is a server-side scripting environment used to combine HTML code with code to interact with COM (Common Object Model) components (of which many are Common Controls). This makes ASP similar to JSP with JSP-s more open approach being the main difference [13]. Therefore, ASP has the same problems as JSP and is not forward compatible.

ASP.NET Master Pages and Web Server Controls

ASP.NET Master Pages [14] is a feature to provide the ability to define common structure and interface elements of a site creating consistent layout throughout the site. This is achieved by using simple templates for grouping site elements present on all site pages.

ASP.NET Web Server Controls [15] are objects on ASP.NET web pages that run when the page is requested and render markup to a browser. The original markup defines common components of a page, but the rendered output depends on the client's capabilities and renderer settings. This creates additional layer of abstraction, which in combination with user controls (embedded ASP.NET pages), makes the solution extendable. When used together with ASP.NET Master Pages, a general templating system is formed.

Similarly to FreeMarker, ASP.NET allows complete separation of presentation and business logic layers. The separation is achieved using code behind and code beside models. Code beside files contain partial classes containing the implementations of web pages events [16]. It is actually the preferred design for building enterprise applications using .NET Framework. Therefore, ASP.NET can be used to create forward compatible presentation layers. However, it is important to note that templates do not ensure forward compatibility.

5.4 Service-Based Software

Since its formation in 1995, DiCE (The Distributed Centre of Excellence in Software Engineering) has been working towards the development of a new approach to the production of highly flexible, but robust, software. In 2000, the group proposed an approach called Service-Based Software [17]. DiCE considered following key issues of future software:

- Software will need to be developed to meet necessary and sufficient requirements. Users should acquire and pay only for the subset of applications features they use.
- Software will be personalized.
- Software will be self-adapting. Software will learn from user actions and try to change in order to better meet user requirements and preferences.
- Software will be fine-grained. Software will be split into independent co-operating components.
- Software will operate in a transparent manner. Software will be seen as one abstract object.

The resulting service-based model of software has following properties:

- Services are configured to meet a specific set of requirements at a point in time, executed and discarded.
- Services are composed out of smaller ones, procured and paid for on demand, as and when needed.
- A service is not a mechanized process. Humans are needed to manage supplier-consumer relationships.

Today, this model is becoming widely accepted as on-demand services are becoming more popular [18]. Service based software also follows the requirements of forward compatible design: it is separated into components (subservices), abstracted into services and subservices, and extensible with configurable features and self-adaption. According to Mullender and Burner, Web Services are used as conceptual level building blocks of enterprise software using service based architecture[19].

Even though service-based software does not tell us how to design forward compatible presentation layer, it does tell us what we should expect from application core (business logic) and how presentation layer should interact with the rest of the application. It tells us that a presentation layer should also be composed of services – different components grouped together and made accessible through message-based interfaces.

6 Requirements for Forward Compatible Presentation Layer Design

Service-based software suggests that a presentation layer should be composed of services – different components grouped together and made accessible through message based interfaces. We noted that Common Controls act and Widgets can act as forward compatible subservices in presentation layer. However, there is no guidance describing, how to design containers in order to achieve forward compatibility in presentation layer. By supplying this guidance, we solve the problem of creating forward compatible presentation layer.

FreeMarker and ASP.NET Master Pages with Web Server Controls allow users to create presentation layers, which are not forward compatible. On the other hand, they both are template-based solutions, which make forward compatible design possible. JSP and ASP as solutions not componentizable into independent components have proven to be not suitable for forward compatible design. Therefore, we can consider only template-based containers and services when designing the framework.

The framework should assure that components created are extensible, can be abstracted and are componentizised as these are the core requirements of forward-compatible design.

Extensibility of template-based containers is not easy to apply, since it needs templates to be flexible. Not all templates allow structural changes to their contents. Luckily, the solutions discussed above do allow structural changes to their contents, which make them extensible. Containers, that allow using templates as components in templates, do inherently allow extensibility.

Abstraction is the requirement the solutions discussed above do not enforce. Abstraction of container level objects is more complex than of the widget level objects. Abstraction requires the ability to consider an object as one system. This means that container level objects in presentation layer need to communicate with business logic layer objects using protocols that allow abstraction and are extensible. That implies using forward compatible protocols like HTTP. Even more, different business logic layer components might require different interfaces with presentation layer components. The old and the new components should work seamlessly together. Solving the communication issue is the key to creating the framework we are looking for.

6.1 Future Proof and Forward Compatible Communications

One solution to communication problems is to have components negotiate their communication needs (protocol format and interfaces) before doing any actual communication. This can cause significant additional load on communication if the components do not bind to each other. Web Services commonly use WSDL to define their communication needs prior to actual communications.

Another solution is to use future-proof and forward compatible communication protocol for communicating. This way the same communication interface can be used with all versions of components.

It is important to keep in mind that the communication interface can be attacked by sending random queries to it. This means that no component can make binding assumptions on data being communicated and should fail gracefully if some needed data is missing or corrupt.

Mario Jeckle and Erik Wilde have suggested[20] that Web Services can be modelled as stack of extendible future-proof layers. In their paper they also pointed out that Web Services also allow queries to subservices, which proves Web Services to be well abstractable and componenizable. This shows that Web Services are close to being forward compatible solutions since Web Services no longer just provide interface for communication as they offer message patterns instead.

6.2 Graceful Degrading of Presentation Layer

Another issue in reaching forward compatibility is graceful degrading when newer technologies come to use. There are several different ways how components degrade.

The easiest way to handle new or unexpected input is to ignore it. This is commonly used when designing XML based communications – the processors will just ignore unknown attributes and elements.

Another option is to respond with error. This solution is commonly used by HTTP servers, which respond with “501 Not Implemented” or “505 HTTP Version Not Supported” to unexpected requests. As responding with error abruptly stops program flow, it might often not be graceful enough. This should be the last solution used.

Ideally, a forward compatible component should try to find out how to handle the new information. The component could ask other components or services for hints on how and whether to handle new information. This allows seamless degradation of components.

It is important to note that the components or services asked for help should avoid giving specific implementations of rendering rules or even the rendering. Trying to give rendering output or detail rules requires that the helping service knows all the services it communicates or will communicate with. This knowledge is important as the actions needed to be taken could be different for different services and versions of service. For example, giving HTML rendering to service rendering for XAML or XUL using client would break the presentation. As forward compatible components should be able to accept even output formats unknown at design time, giving detail instructions or renderings breaks forward compatibility.

There are several different approaches when asking how to handle new or unknown input.

Service Based Approach

The first solution is to use a dedicated service that tries to find out what to do with new input. This solution follows Service Based Architecture the best. In fact, the service itself could handle new input by asking itself for directions.

The implementation of such service could be either configuration driven or just routing based.

Configuration Driven Implementation

Configuration driven implementation means that all responses are read from service configuration database (figure 2). This means that for any update that affects communications between forward compatible services, a new configuration record has to be added to the database in order to allow truly seamless upgrades. However, keeping configuration up to date can be difficult to implement.

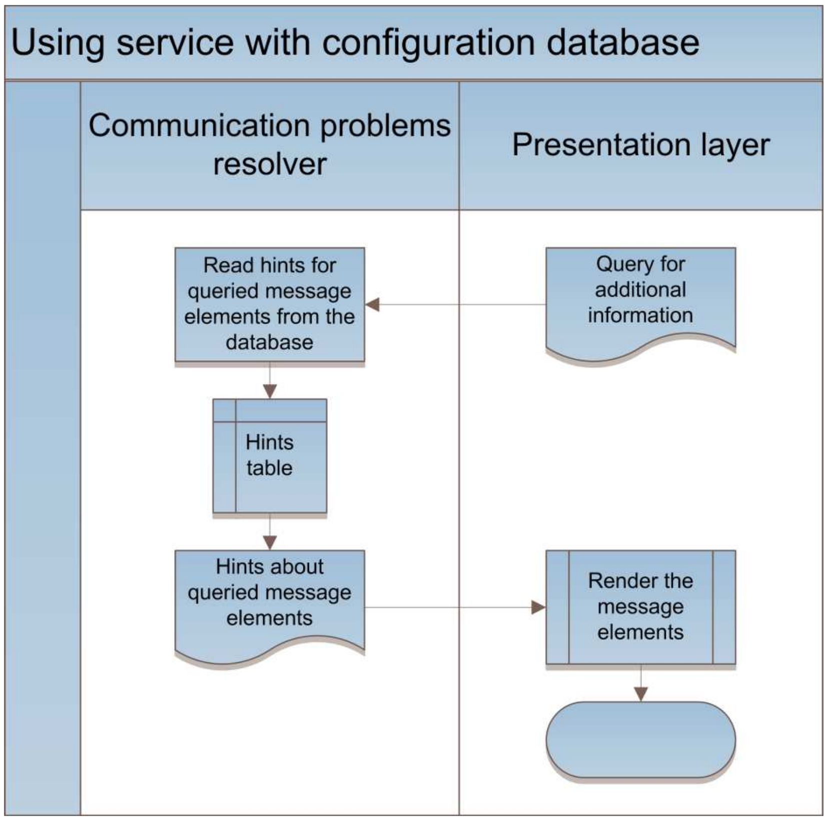
Using service with configuration database
Communication problems resolver
Presentation layer
Read hints for queried message elements from the database
Query for additional information
Hints table
Hints about queried message elements
Render the message elements

Figure 2. Service based solution using configuration database.

Routing Based Implementation

Routing based implementation routes queries to services responsible for storing the rules for the type of messages received by the querying service (figure 3).

Often the responsible service could be the same that sent the original message. In this case, all the services need to support queries for processing directions, which makes their design and implementation more complex. Therefore, services sending the messages should avoid the need for making queries how the messages should be handled. For example the messages could be designed to contain semantic information that could be used during rendering process.

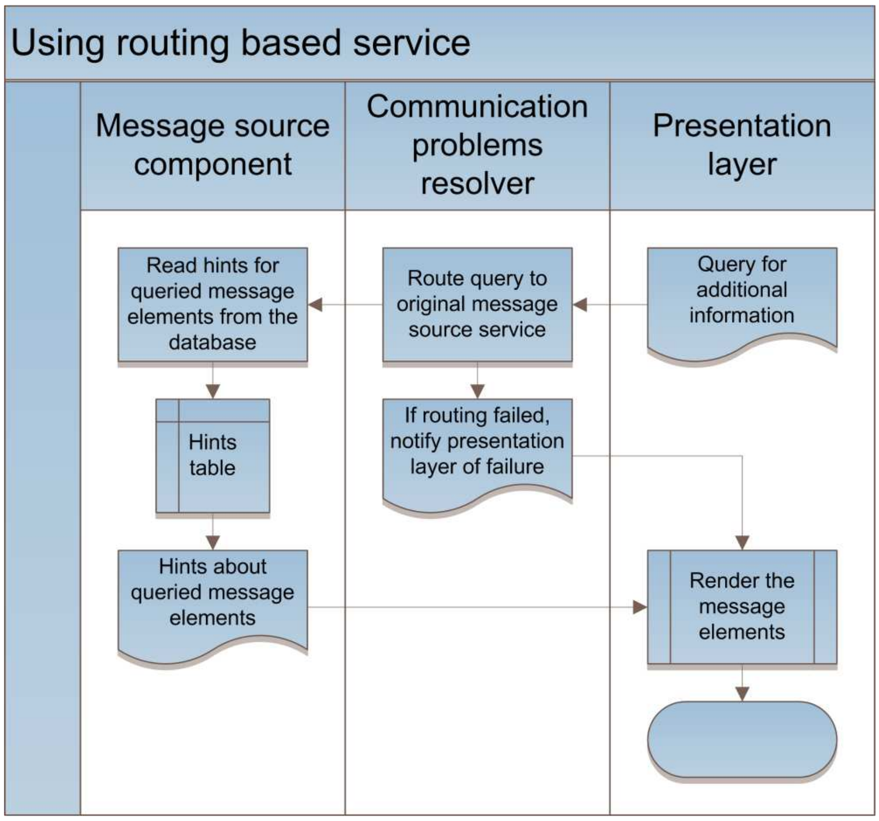
Using routing based service
Message source component
Communication problems resolver
Presentation layer
Read hints for queried message elements from the database
Route query to original message source service
Query for additional information
Hints table
If routing failed, notify presentation layer of failure
Hints about queried message elements
Render the message elements

Figure 3. Service based solution using routing.

Mixed Implementation

In order to reduce complexity of services responding to “how to handle messages” queries, a mixture of routing based and configuration driven services can be used (figure 4). Routing based services can be used to filter queries according to message types. Filtered queries could then be sent to services specialized on the corresponding types of messages (message semantics providers). This method is most effective if multiple components use messages with similar information.

Mixed implementation and routing based implementation allow using third party message translators to be used. Configuration based solution does not provide the option to use third party services for translating messages.

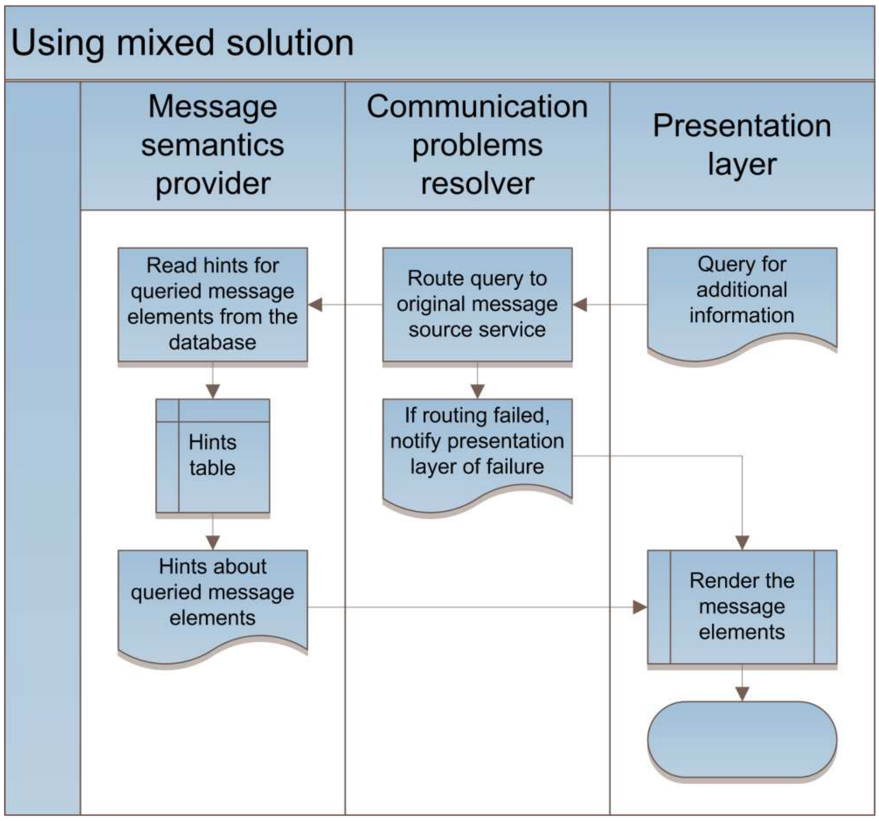
Using mixed solution
Message semantics provider
Communication problems resolver
Presentation layer
Read hints for queried message elements from the database
Route query to original message source service
Query for additional information
Hints table
If routing failed, notify presentation layer of failure
Hints about queried message elements
Render the message elements

Figure 4. Service based solution using mixed approach.

Internal Configuration Approach

The second solution to handling the input is using the needed configuration internally in the receiving service. This is actually used often in services today.

It could be a good solution if there were only a few services interacting with each other. However, it may raise a problem if there are many services whose input gets affected by updates in the service they communicate with. Internal configuration approach requires updates to the configurations of all the services affected by the changes in communication protocols. Consequently, some services can be overlooked causing these to fail at some point.

If the configurations were stored in one service, the service could be used to inform developers about which services are affected by the changes. In this case, the affected services are harder to overlook. In addition, the same configuration entry could be used for similar services lowering the need to write duplicate configurations and making it easier to fix configuration errors.

Subservice Approach

The third option would be using an internal subservice in sending component to solve communication issues.

Using the subservice is very similar to using dedicated service (figure 5). The main difference is that using subservice requires subservice for all components while one dedicated service could be used to serve all affected services.

The subservice could also act as an upgrade to using internal solution. Compared to internal configuration the service could take advantage of additional semantic information queried from the sender as proposed above. Subservice approach allows using third party message translators.

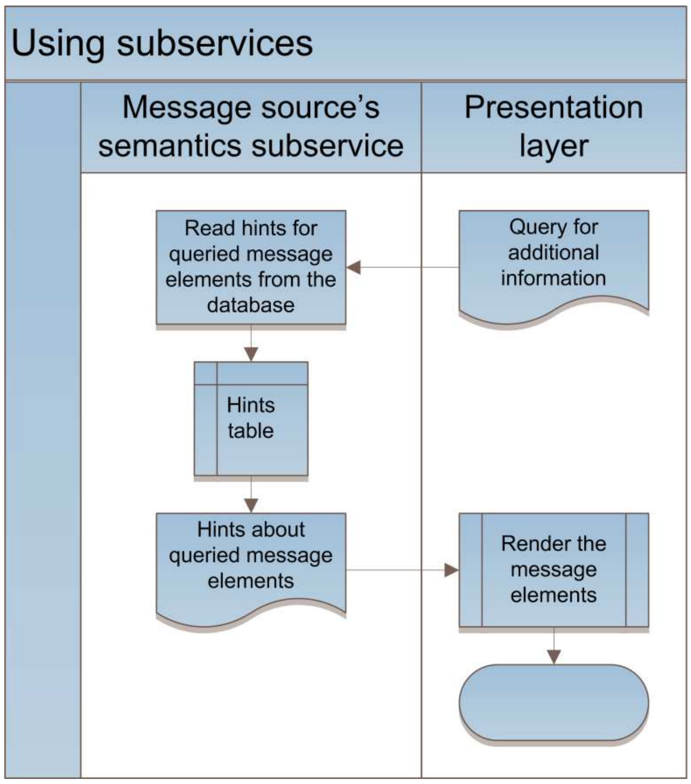
Using subservices
Message source's semantics subservice
Presentation layer
Read hints for queried message elements from the database
Query for additional information
Hints table
Hints about queried message elements
Render the message elements

Figure 5. Subservices based solution.

Monodirectional solutions

In some cases the presentation level cannot automatically send out queries on how to handle input from business logic layer. This might be an issue when using XSLT as presentation layer solution. It might be possible to get some feedback using XML processor instructions in XSLT if the XSLT processor used supports these. Unfortunately many XSLT processors do not support processor instructions from XSLT files.

If the presentation layer cannot ask for advice on how to handle input, it will no longer be fully forward compatible since some information has to retain its format. For example, a component that used to get input in EDIFACT format[21] cannot work with XML format unless it has means to ask how to interpret it. However, we can provide some forward compatibility as long as the structural format remains the same. We can allow replacements, deletions and additions of data blocks in incoming transmissions.

Deletions can be enabled by permitting some parts of the message to be missing. In fact, because different components render different blocks of messages, we can just ignore components not getting input.

Insertion can be handled if inserted items carry some additional information. For example, they could include some semantics that can give hints to presentation layer about which method of rendering might be appropriate. If XML is used for communication between presentation and business logic layer, then that extra information could be stored in attributes of added data elements. It is also worthwhile to notice that well designed XML document element and attribute names already carry some semantic information in order to be understandable to humans. If additional attributes or elements are used for giving additional semantic information, the hinting attributes or elements should use namespace different from other namespaces used in document. That way they will not interfere with message information.

Replacement can be considered as deletion and insertion of new item. Therefore it needs no further consideration.

7 Guidelines for Forward Compatible Presentation Layers

7.1 Guidelines

Having evaluated present solutions, we provide guidelines on how web services presentation layers should be designed for forward compatibility. It is important to follow forward compatibility through all presentation layers. Applications do not just become easier to upgrade, forward compatibility also increases component reuse.

When designing a new presentation layer, following guidelines should be considered:

1. Use Common Controls or Widgets when possible. This avoids duplicate code and gives forward compatible controls layer.
2. Combine container layer objects and components layer objects to create new container layer objects that inherit forward compatibility from its components. Try to reuse the new container layer objects.
3. Use general services rather than internal components to expose similar aspects of objects or types of objects. This way you can combine the strengths of aspect oriented programming (AOP)[22] with the strengths of object oriented programming (OOP).
4. Compose complex services out of individually addressable and subscribable subservices. This lowers the communications overhead and follows the newest Web Services standards.
5. Avoid the need for request for additional information by the presentation layer. Try to maximize the use of semantic information already present in communications protocols. This way your presentation layer becomes intelligent, degrades gracefully and gains generality, which can be used when designing new features to present solution.
6. Provide means to request additional semantic information about objects used in messages from business layer objects. Supply default actions for cases where presentation layer needs more information than can be supplied.

7.2 Enforcing the Guidelines

In order to maximize the benefits of forward compatibility, means to automatically enforce following of these guidelines, should be put into use. Next we are going to consider, what can be done to enforce the guidelines.

1. First guideline can be enforced by removing the ability to create custom basic controls. This may, however, result in lower performance of the solution as some simple tasks might have to be addressed by complex components. In some cases, widgets or common controls might not be available due to the uniqueness of the solution. In that case, a new set of common building blocks needs to be created.
2. The second guideline can only be enforced in conjunction with the first guideline. In fact, by denying the ability to create new components, we also force them to create new objects by combining present objects. It is difficult to enforce the reuse, however, it is possible to detect similar portions of code and present warnings about these at compile time.
3. The third guideline is difficult, if not impossible, to enforce automatically. Detection of similar code portions can be used here as well. However, it is very difficult to automatically detect whether two code pieces express the same aspects.
4. There is no distinct metric that could be used to decide whether a service should be divided into subservices. It is still possible to use number of different message types or contexts as an approximal metric. This metric can be used to display compile time warnings at chosen value ranges.
5. By removing the ability to ask for more information, we can enforce the fifth guideline. However, that way we also lose the ability to reach full forward compatibility as discussed above. Alternatively we can define minimal sets of semantic information that has to be supplied with message elements. This way we still allow requesting additional information, however, the minimal set might not be satisfactory in all cases. The presence of the minimal semantic information can be verified automatically.
6. It is possible to require all new components to implement interface for querying for additional information about objects. The implementation of the interface can be verified automatically.

As seen above, it is impossible to automatically enforce all the guidelines presented. This leaves most conformances checking to human. It is common to use patterns and best practices documents to establish common knowledge about guidelines that are difficult to enforce. Still, automatic verification can be used with some guidelines. Probable deviations from most guidelines can be highlighted out with compile time warnings.

7.3 Notes about Guidelines for Software Product-Line Evolution

M. Svahnberg and J. Bosch give guidelines for software product-line evolution[4]. The given guidelines are outlined as follows.

1. Avoid adjusting component interfaces. Sooner or later the interfaces of components will need to be changed.
2. Focus on making component interfaces general.
3. Separate domain and application behaviour.
4. Keep the software product line intact.
5. Detect and exploit common functionality in component implementations.
6. Be open to rewriting components and implementations.
7. Avoid hidden assumptions and design decisions in the source code.

These guidelines were to minimize the cost of software evolution. It is interesting to see that forward compatible design is following many of these guidelines. As forward compatible components use future proof interfaces, the interfaces will be general. Also, ability to accept new input messages avoids adjusting these interfaces and encourages rewriting of components and services without breaking the product line. It also makes the components open to any input, forcing the components to avoid hidden assumptions, because these would break forward compatibility. Therefore, it can be assumed that forward compatibility lowers the cost of software evolution.

8 Example of Forward Compatible Design

8.1 XSLT based design

Background

In VabaVaraVeeb users can create their own presentation of portal. Even though all the styles created by users are currently overseen by portal administrators before allowed to be put to use, work towards automating and supporting administrators to verify styles conformance to portals policies, is in progress. For example, requirements to user designed styles can now be specified using xslt-req, which allows automatic verification of many aspects usually verified by manual review. It is interesting to note that the styles are not limited to displaying HTML or RSS, but support any format user wants to use. This makes the solution ready for new technologies like using XAML to display the information in windows application like interface. This kind of preparedness to new technologies is a good basis for forward compatible presentation layer.

The problem with user created custom styles is that users rarely take time to update their styles to reflect changes in the portal services. This makes styles short-living and discourages users to create new styles. The solution to the problem can be seen in enforcing forward compatible design of styles as much as possible.

The styles consist of XSL transformations and supporting files (images, css, JavaScript, etc.). In fact, the whole presentation layer is made of XSL transformations that can be applied either at the server or in client application. The portal currently uses XML to represent information in its business layer.

Design

The guidelines were put into use when designing a new presentation layer for VabaVaraVeeb. The guidelines were addressed with the following techniques.

Provided Fallback to Defaults

In order to allow easy modification of styles, all styles created by users are required to import the portals default style. This allows them to override presentation details only in locations they are interested in. The default style will be used to handle the aspects and information the user does not want to present in a different way. It also follows guidelines

1 and 2 giving users access to controls and templates in the default style. An example of forward compatible custom template is shown in listing 1.

```
<!--Tree of checkboxes -->
<xsl:template name="t_valikute_puu">
   <!-- The root node of the tree -->
   <xsl:param name="juur" select="."/>
   <!-- The maximum depth of the tree -->
   <xsl:param name="max_sygavus" select="3"/>
   <!-- Prefix to differenciate different trees -->
   <xsl:param name="nime_prefiks"/>
   <!-- Create HTML block to contain the tree -->
   <div class="tree">
      <xsl:call-template name="t_valikute_alampuu" >
         <xsl:with-param name="juur" select="$juur"/>
         <xsl:with-param name="max_sygavus" select="$max_sygavus"/>
         <xsl:with-param name="nime_prefiks" select="$nime_prefiks"/>
      </xsl:call-template>
   </div>
</xsl:template>
<!--Subtree of options -->
<xsl:template name="t_valikute_alampuu">
   <!-- Current nodes id -->
   <xsl:param name="id" select="0"/>
   <!-- Current node (root of subtree) -->
   <xsl:param name="juur" select="."/>
   <!-- Depth of branch -->
   <xsl:param name="sygavus" select="0"/>
   <!-- Maximum depth of tree -->
   <xsl:param name="max_sygavus" select="3"/>
   <!-- Prefix to differenciate different trees -->
   <xsl:param name="nime_prefiks"/>
   <!-- Left margin of subtree block (1 unless it is the root of tree) -->
   <xsl:variable name="margin">
      <xsl:choose>
         <xsl:when test="$sygavus &gt; 0">1</xsl:when>
         <xsl:otherwise>0</xsl:otherwise>
      </xsl:choose>
   </xsl:variable>
   <!-- Subtree container -->
   <div class="subtree" style="margin-left:{$margin}em;">
      <!-- For each child, display it -->
      <xsl:for-each select="$juur/../*[ylem_id = $id and name(.) = name($juur)]">
         <xsl:variable name="this_id" select="./@id"/>
         <!-- Whether user should be able to expand the node -->
         <xsl:variable name="this_expand" select="$sygavus &lt; $max_sygavus and
count(../*[ylem_id = $this_id  and name(.) = name($juur)]) &gt; 0"/>
         <!-- If the node is expandable, add event handlers and show appropriate
icon -->
         <xsl:choose>
            <xsl:when test="$this_expand = 'true'">
               <img src="graafika/silk013/icons/bullet_toggle_minus.png"
alt="peida alampuu" onclick="javascript:changeSubTreeVisibility(this);"
onkeypress="javascript:changeSubTreeVisibility(this);" style="cursor: pointer;"
/>
            </xsl:when>
            <xsl:otherwise>
               <img src="graafika/silk013/icons/bullet_white.png" alt="leht"/>
            </xsl:otherwise>
         </xsl:choose>
         <!-- Create checkbox and add event handlers -->
         <input type="checkbox" name="{$nime_prefiks}_id_{./@id}"
id="{$nime_prefiks}_id_{$id}_{./@id}" value="{./@id}"
onclick="javascript:setInputCheck(this.name, this.checked);"
onkeypress="javascript:setInputCheck(this.name, this.checked);">
            <!-- Check chosen options -->
            <xsl:if test="./@valitud = 1">
               <xsl:attribute name="checked">checked</xsl:attribute>
            </xsl:if>
         </input>
         <!-- Label the checkbox -->
         <label for="{$nime_prefiks}_id_{$id}_{./@id}" title="{./@kirjeldus}">
            <xsl:value-of select="./@nimi"/>
         </label>
         <br/>
         <!-- Output current nodes subtree -->
         <xsl:if test="$this_expand = 'true'">
            <xsl:call-template name="t_valikute_alampuu">
               <xsl:with-param name="id" select="./@id"/>
               <xsl:with-param name="sygavus" select="$sygavus + 1"/>
               <xsl:with-param name="juur" select="."/>
               <xsl:with-param name="max_sygavus" select="$max_sygavus"/>
               <xsl:with-param name="nime_prefiks" select="$nime_prefiks"/>
            </xsl:call-template>
         </xsl:if>
      </xsl:for-each>
   </div>
</xsl:template>
```

Listing 1. Template that generates a tree of records with checkboxes in HTML format.

Tree of records with checkboxes represents graph by choosing one node (root) and drawing its children up to preset depth (given with *max_sygavus*). As one and the same item can be represented more than once, JavaScript is used to synchronize the copies (invoked at *onclick* or *onkeypressed* events). Checkbox names and ids are prefixed with *nime_prefix* in order to differentiate different trees on page. The template is invoked as shown on the following example in listing 2.

```
<!-- ülemkategooriad -->
<fieldset title="Kategooriates" id="tree">
   <xsl:call-template name="t_valikute_puu">
      <xsl:with-param name="juur" select="./kategooriad/kategooria[1]"/>
      <xsl:with-param name="nime_prefiks">ylemkategooria</xsl:with-param>
   </xsl:call-template>
</fieldset>
```

Listing 2. Example of using custom component to display tree of categories.

The system represents the graphs by listing all its nodes with their identifier (attribute *id*), name (attribute *nimi*), description (attribute *description*), flag showing whether the node is selected (attribute *valitud*) and a list of the nodes parents identifiers (elements *ylem_id*). A sample fragment of XML that can be rendered as a tree using previously listed code is shown in listing 3. The result of applying the template can be seen in listing 4. Supporting JavaScript code used for synchronization is in the appendix.

```
<kategooriad>
   <kategooria id="1" nimi="A" kirjeldus="A" valitud="0">
      <ylem_id>0</ylem_id>
   </kategooria>
   <kategooria id="2" nimi="B" kirjeldus="B" valitud="1">
      <ylem_id>0</ylem_id>
   </kategooria>
   <kategooria id="3" nimi="A1" kirjeldus="A_1" valitud="0">
      <ylem_id>1</ylem_id>
   </kategooria>
   <kategooria id="4" nimi="B1" kirjeldus="B1" valitud="0">
      <ylem_id>2</ylem_id>
   </kategooria>
   <kategooria id="5" nimi="C/A2a" kirjeldus="C/A2a" valitud="1">
      <ylem_id>0</ylem_id>
      <ylem_id>6</ylem_id>
   </kategooria>
   <kategooria id="6" nimi="A2" kirjeldus="A2" valitud="0">
      <ylem_id>1</ylem_id>
   </kategooria>
   <kategooria id="7" nimi="A1a/B2" kirjeldus="A1a/B2" valitud="0">
      <ylem_id>3</ylem_id>
      <ylem_id>2</ylem_id>
   </kategooria>
   <kategooria id="8" nimi="A2b" kirjeldus="" valitud="0">
      <ylem_id>6</ylem_id>
   </kategooria>
</kategooriad>
```

Listing 3. Sample fragment of XML to be rendered as a tree.

```
<fieldset title="Kategooriates" id="tree">
   <div class="tree">
      <div class="subtree" style="margin-left:0em;">
         <img src="graafika/silk013/icons/bullet_toggle_minus.png" alt="peida
alampuu" onclick="javascript:changeSubTreeVisibility(this);"
onkeypress="javascript:changeSubTreeVisibility(this);" style="cursor: pointer;"
/>
         <input type="checkbox" name="ylemkategooria_id_1"
id="ylemkategooria_id_0_1" value="1"
onclick="javascript:setInputCheck(this.name, this.checked);"
onkeypress="javascript:setInputCheck(this.name, this.checked);" />
         <label for="ylemkategooria_id_0_1" title="A">A</label>
         <br />
         <div class="subtree" style="margin-left:1em;">
            <img xmlns="" src="graafika/silk013/icons/bullet_toggle_minus.png"
alt="peida alampuu" onclick="javascript:changeSubTreeVisibility(this);"
onkeypress="javascript:changeSubTreeVisibility(this);" style="cursor: pointer;"
/>
            <input type="checkbox" name="ylemkategooria_id_3"
id="ylemkategooria_id_1_3" value="3"
onclick="javascript:setInputCheck(this.name, this.checked);"
onkeypress="javascript:setInputCheck(this.name, this.checked);" />
            <label for="ylemkategooria_id_1_3" title="A_1">A1</label>
            <br />
            <div class="subtree" style="margin-left:1em;">
               <img src="graafika/silk013/icons/bullet_white.png" alt="leht" />
               <input type="checkbox" name="ylemkategooria_id_7"
id="ylemkategooria_id_3_7" value="7"
onclick="javascript:setInputCheck(this.name, this.checked);"
onkeypress="javascript:setInputCheck(this.name, this.checked);" />
               <label for="ylemkategooria_id_3_7" title="A1a/B2">A1a/B2</label>
               <br />
            </div>
            <img src="graafika/silk013/icons/bullet_toggle_minus.png" alt="peida
alampuu" onclick="javascript:changeSubTreeVisibility(this);"
onkeypress="javascript:changeSubTreeVisibility(this);" style="cursor: pointer;"
/>
            <input type="checkbox" name="ylemkategooria_id_6"
id="ylemkategooria_id_1_6" value="6"
onclick="javascript:setInputCheck(this.name, this.checked);"
onkeypress="javascript:setInputCheck(this.name, this.checked);" />
            <label for="ylemkategooria_id_1_6" title="A2">A2</label>
            <br />
            <div class="subtree" style="margin-left:1em;">
               <img src="graafika/silk013/icons/bullet_white.png" alt="leht" />
               <input type="checkbox" name="ylemkategooria_id_5"
id="ylemkategooria_id_6_5" value="5"
```

```
onclick="javascript:setInputCheck(this.name, this.checked);"
onkeypress="javascript:setInputCheck(this.name, this.checked);"
checked="checked" />
          <label for="ylemkategooria_id_6_5" title="C/A2a">C/A2a</label>
          <br />
          <img src="graafika/silk013/icons/bullet_white.png" alt="leht" />
          <input type="checkbox" name="ylemkategooria_id_8"
id="ylemkategooria_id_6_8" value="8"
onclick="javascript:setInputCheck(this.name, this.checked);"
onkeypress="javascript:setInputCheck(this.name, this.checked);" />
          <label for="ylemkategooria_id_6_8" title="">A2b</label>
          <br />
        </div>
      </div>
      <img src="graafika/silk013/icons/bullet_toggle_minus.png" alt="peida
alampuu" onclick="javascript:changeSubTreeVisibility(this);"
onkeypress="javascript:changeSubTreeVisibility(this);" style="cursor: pointer;"
/>
      <input type="checkbox" name="ylemkategooria_id_2"
id="ylemkategooria_id_0_2" value="2"
onclick="javascript:setInputCheck(this.name, this.checked);"
onkeypress="javascript:setInputCheck(this.name, this.checked);"
checked="checked" />
      <label for="ylemkategooria_id_0_2" title="B">B</label>
      <br />
      <div class="subtree" style="margin-left:1em;">
        <img src="graafika/silk013/icons/bullet_white.png" alt="leht" />
        <input type="checkbox" name="ylemkategooria_id_4"
id="ylemkategooria_id_2_4" value="4"
onclick="javascript:setInputCheck(this.name, this.checked);"
onkeypress="javascript:setInputCheck(this.name, this.checked);" />
        <label for="ylemkategooria_id_2_4" title="B1">B1</label>
        <br />
        <img src="graafika/silk013/icons/bullet_white.png" alt="leht" />
        <input type="checkbox" name="ylemkategooria_id_7"
id="ylemkategooria_id_2_7" value="7"
onclick="javascript:setInputCheck(this.name, this.checked);"
onkeypress="javascript:setInputCheck(this.name, this.checked);" />
        <label for="ylemkategooria_id_2_7" title="A1a/B2">A1a/B2</label>
        <br />
      </div>
      <img src="graafika/silk013/icons/bullet_white.png" alt="leht" />
      <input type="checkbox" name="ylemkategooria_id_5"
id="ylemkategooria_id_0_5" value="5"
onclick="javascript:setInputCheck(this.name, this.checked);"
onkeypress="javascript:setInputCheck(this.name, this.checked);"
checked="checked" />
      <label for="ylemkategooria_id_0_5" title="C/A2a">C/A2a</label>
      <br />
    </div>
  </div>
</fieldset>
```

Listing 4. Rendering of sample XML fragment.

Use xslt-req To Add Additional Semantics

xslt-req is used to give extensibility hints. xslt-req used together with XML Schema tells style designer, which XML elements or attributes are allowed to change and whether the styles should bother presenting the information given by these elements. Even though having some of the message structure fixed brakes forward compatibility, it adds some more semantics to the messages. That additional semantics can be used to render new services or components not known at design time. This follows guideline 5.

Use Semantics Already Provided By the Business Layer

Default style is designed in a way it makes use of semantic information already present in the incoming XML document. For example it uses the fact that the inner representation XML is using mainly Estonian element names to detect possible lists of components. In Estonian multiples generally have a letter ‘d’ at the end of the word (similarly English words representing multiples have ‘s’ as the last letter). This knowledge can be used to assume elements with names ending with ‘d’ are probably lists of items. This follows guideline 5.

This kind of semantic information does not always give us the correct interpretation. Even by improving the list detection by assuming certain attributes to be present on lists, might cause misinterpretation. In fact, semantic detections are rarely always correct – even humans may misinterpret the information. Fortunately, the misinterpretations are detectable by manual verification and messages can be redesigned to be clearer to avoid misinterpretation.

```
<!-- List container -->
<!-- Matches all elements ending with 'd' -->
<xsl:template name="t_komponent_loend"
      match="*[substring(name(.), string-length(name(.))) = 'd']">
   <xsl:param name="konteiner" select="."/>
   <!-- Container block -->
   <div class="kast">
      <!-- Header -->
      <div class="top">
         <xsl:if test="$konteiner/@nimi and string-length($konteiner/@nimi) &gt;
0">
         <!-- Show help icon on the top right corner -->
            <span class="topright">
               <xsl:call-template name="abi">
                  <xsl:with-param name="teema" select="$konteiner/@nimi" />
               </xsl:call-template>
            </span>
         </xsl:if>
         <!-- Display header texts -->
         <span>
            <!-- List name -->
            <xsl:value-of select="$konteiner/@nimi"/>
            <!-- Range of items shown and total items -->
            <xsl:if test="$konteiner/@kokku">
               <xsl:text> (</xsl:text>
               <xsl:if test="$konteiner/@esimene">
                  <xsl:value-of select="$konteiner/@esimene"/>
                  <xsl:text> - </xsl:text>
                  <xsl:value-of select="$konteiner/@viimane"/>
                  <xsl:text> / </xsl:text>
               </xsl:if>
               <xsl:value-of select="$konteiner/@kokku"/>
               <xsl:text>)</xsl:text>
            </xsl:if>
         </span>
      </div>
      <!-- Display container items -->
      <div class="sisu">
         <xsl:apply-templates select="$konteiner/*"/>
      </div>
      <!-- Display navigation for next and previous page if present -->
      <div class="top" align="center">
         <form action="index.php?Otsing=" method="post" class="postbackform">
            <input type="hidden" name="otsing_pide" value="{$konteiner/@pide}"/>
            <xsl:if test="./@lk &gt; 1">
               <a href="index.php?Otsing=lk={$konteiner/@lk -
1}&amp;otsing_pide={$konteiner/@pide}" title="eelmine lk">
                  <img src="graafika/silk013/icons/book_previous.png"
                        alt="eelmine lk" class="ikoon"/>
               </a>
            </xsl:if>
            <xsl:text> | </xsl:text>
            <xsl:if test="./@kokku &gt; ./@viimane">
               <a href="index.php?Otsing=lk={$konteiner/@lk +
1}&amp;otsing_pide={$konteiner/@pide}" title="järgmine lk">
                  <img src="graafika/silk013/icons/book_next.png" alt="järgmine lk"
class="ikoon"/>
               </a>
            </xsl:if>
         </form>
      </div>
   </div>
</xsl:template>
```

Listing 5. Template to generate listing of items using element names ending with 'd' to detect lists.

Templates Are Separated Into Different Mostly Independent Files

Templates are separated based on their usage. This means that templates handling similar aspects are grouped together, just as templates used for presenting different subservices or modules are separated. For example, custom controls used to present similar aspects are in one file; module specific rendering is done using specialized stylesheet files. For example, listing container template is in components stylesheet while application module specific rendering of list items is done in application module stylesheet. This follows guideline 3.

Services to Provide Additional Semantical Information Are Available

A new service was written to allow styles ask for more semantic information. The styles could invoke the service using AJAX during rendering time or by using AJAX or any other method to post a HTTP request. This follows guideline 6.

Services Are Individually Addressable and Subscribable

With the aim of making services individually addressable and subscribable, a new service infrastructure was created. The new infrastructure uses service manager module to route requests to the services requested. It also manages service requests between business layer services. This reduces the network load as requests between business layer services are handled internally opposed to making all requests uniformly through the network interface. This infrastructure allows addressing of any services or subservices in the solution following guideline 4. The service manager component also provides the service level support for presentation layer by choosing the presentation templates according to user preferences and client capabilities.

Furthermore, services to handle similar aspects in different modules were made available. This follows guideline 3.

Graceful Degrading Of Components

A mixed solution for graceful degrading of components was chosen. First, specific rules are looked for, then semantic information about input is used to find templates. This kind of behavior is achieved by adding priorities to templates. If there is still not enough information to render the data, AJAX or ordinary HTTP request can be used to ask the business logic layer for more information. If not enough information is available, default action given by the request for more information is used. If the response does not give default action, the element is ignored. The rendering process is described on figure 6. This follows guidelines 5 and 6.

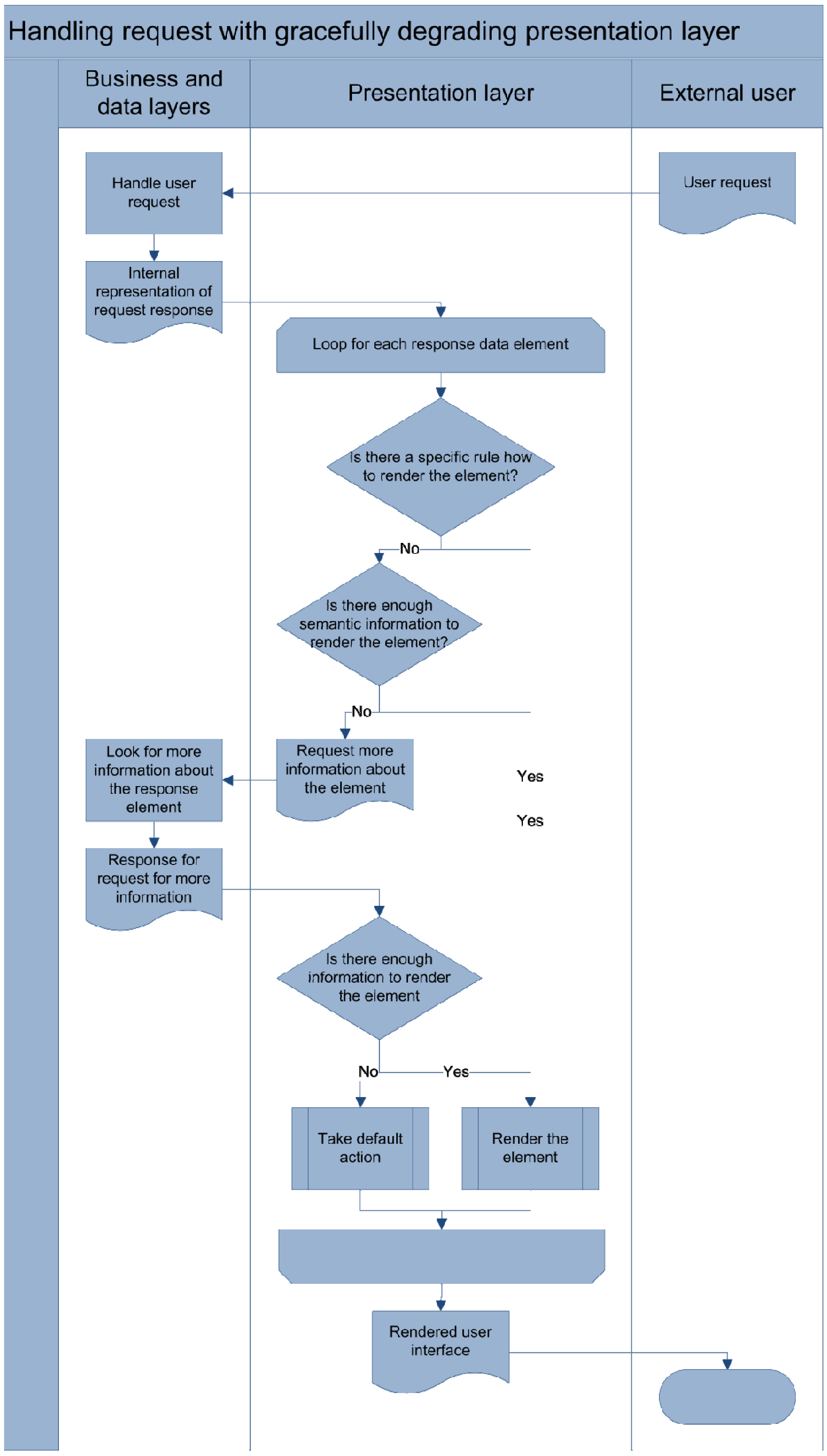
Handling request with gracefully degrading presentation layer
Business and data layers
Presentation layer
External user
User request
Handle user request
Internal representation of request response
Loop for each response data element
Is there a specific rule how to render the element?
No
Is there enough semantic information to render the element?
No
Request more information about the element
Yes
Yes
Look for more information about the response element
Response for request for more information
Is there enough information to render the element
No
Yes
Take default action
Render the element
Rendered user interface

Figure 6. Graceful degrading of presentation layer components in VabaVaraVeeb.

Results

The new design made it possible to use similar presentation for all modules in the solution without the need to write specialized presentation layer code for each of these. Because the data was internally stored in XML format, the components did not need much of rewriting as much of the old XML structure could be used. It was expected benefit as the original XML structure was used as the basis during designing new presentation layer.

Adding new components has also become simpler. If new components are internally represented following the same pattern as old ones, they do not need any additional work to update the presentation layer since the presentation layer knows how to handle the new components using hints in the components representation. For more complex components, an additional hinting subservice can be created. It is important to note that the subservice can only be in hinting role, not presenting role, as the format of resulting output might be unknown.

When users design their new styles, XML Schema and xslt-req defined base of messages is used to verify their correctness. It does not grant perfect verification, but still automates some of the verification used to be done manually. For example, verifying whether certain values or elements are present or missing in the output can be automated. It is also possible to use automatic verification to confirm that the new styles are extendable where needed and specified by the XML Schema.

9 Conclusions

As seen from the example, forward compatible presentation layer requires significant support from the other functionality layers. In fact, forward compatible presentation layer requires the communication with business layer to be forward compatible as well. This, in turn, benefits from service based architecture of business layer.

Even though creating forward compatible presentation layer requires more code to be written in business logic layer to allow hinting or translating services, adding new components requires less presentation layer code to be written. This can be extremely useful if the number of services using similar interfaces is high. Well designed presentation layer will be able to support large number of additional software features before requiring any changes at all. The latter can be considered a benefit of service based software.

The example followed all six proposed guidelines.

9.1 Future work

In this thesis only forward compatible presentation layer is discussed. Presentation layer, however, is one of many layers in modern software. Most of these would probably benefit from forward compatibility. All these layers have their specific problems that need to be addressed separately. Researching ways to make other software layers forward compatible and how to effectively use these together, will be a potential course of future work.

On the other hand, the aim of allowing users design their own interfaces to be used with software, can be pursued. This means improving automatic verification of user designed interfaces and creating easy to use intuitive means of editing the styles. For example, automatic verification can be improved by adding element and attribute name patterns support to XML Schema and xslt-req.

10 References

[1] **World Wide Web Consortium ,** “XSL Transformations (XSLT),” [Online] 16 November 1999. [Cited: 24 June 2006.] http://www.w3.org/TR/xslt.

[2] **S. Karus,** "Kasutajate poolt loodud XSL teisendustele esitavate nõuete spetsifitseerimine," Faculty Of Mathematics And Computer Science, University of Tartu. Tartu, 2005. Bachelors Thesis.

[3] **C. Armbruster,** "DESIGN FOR EVOLUTION - An Internet White paper on Forward Compatible Design," *http://chrisarmbruster.com/.* [Online] 1999. [Cited: June 22, 2006.] Web Week '98 presentation (Long Branch, NJ). http://chrisarmbruster.com/documents/D4E/witepapr.htm.

[4] **M. Svahnberg and J. Bosch,** "Evolution in software product lines: two cases," *Journal of Software Maintenance: Research and Practice*, Vol. 11, no. 6, pp. 391-422, s.l. : John Wiley & Sons, Ltd., December 21, 1999. DOI: 10.1002/(SICI)1096-908X(199911/12)11:6<391::AID-SMR199>3.0.CO;2-8.

[5] **E. Christensen, F. Curbera, G. Meredith and S. Weerawarana,** "Web Services Description Language (WSDL) 1.1," *World Wide Web Consortium.* [Online] Ariba; International Business Machines Corporation; Microsoft, March 15, 2001. [Cited: January 30, 2007.] http://www.w3.org/TR/wsdl.

[6] **Microsoft Corporation,** "Windows Controls," *Microsoft Developers Network Library.* [Online] [Cited: September 24, 2006.] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/shellcc/platform/commctls/common/common.asp.

[7] **B. McKinney,** "Famous Explorers and Common Controls," *Hardcore Visual Basic.* Redmond : Microsoft Press, 1997. Available at: http://vb.mvps.org/hardcore/index.html.

[8] **D. Howe,** "Free On-line Dictionary of Computing," [Online] 1993. [Cited: September 24, 2006.] http://foldoc.org/.

[9] **Sun Microsystems, Inc.** "JavaBeans," *Sun Developer Network.* [Online] [Cited: September 24, 2006.] http://java.sun.com/products/javabeans/index.jsp.

[10] **Sun Microsystems, Inc,** "JavaServer Pages Technology," *Sun Developer Network* . [Online] Sun Microsystems, Inc. [Cited: 10 6, 2006.] http://java.sun.com/products/jsp/.

[11] **V. DiBartolo,** "FreeMarker: An open alternative to JSP," *JavaWorld.com.* [Online] January 2001. [Cited: October 6, 2006.] http://www.javaworld.com/javaworld/jw-01-2001/jw-0119-freemarker.html?.

[12] **Microsoft Corporation,** "Active Server Pages," *MSDN Library* . [Online] Microsoft Corporation. [Cited: October 6, 2006.] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnanchor/html/activeservpages.asp.

[13] **Sun Microsystems, Inc. ,** "JavaServer Pages[tm] Technology - Comparison with ASP," *Sun Developer Network* . [Online] Sun Microsystems, Inc. . [Cited: October 6, 2006.] http://java.sun.com/products/jsp/jsp-asp.html.

[14] **Microsoft Corporation,** "ASP.NET Master Pages ," *MSDN Library.* [Online] Microsoft Corporation. [Cited: October 6, 2006.] http://windowssdk.msdn.microsoft.com/en-gb/library/18sc7456.aspx.

[15] **Microsoft Corporation,** "ASP.NET Web Server Controls," *MSDN Library.* [Online] Microsoft Corporation. [Cited: October 6, 2006.] http://windowssdk.msdn.microsoft.com/en-gb/library/7698y1f0.aspx.

[16] **Microsoft Corporation,** "INFO: ASP.NET Code-Behind Model Overview," *Microsoft Support Knowledge Base.* [Online] February 23, 2007. [Cited: March 28, 2007.] http://support.microsoft.com/kb/303247. Q303247.

[17] **K. Bennett, P. Layzell, D. Budgen, P. Brereton, L. Macaulay and M. Munro,** "Service-Based Software: The Future for Flexible Software," *Software Engineering Conference, 2000. APSEC 2000. Proceedings. Seventh Asia-Pacific.*Singapore : IEEE, 2000, pp. 214-221. Available at: http://www.bds.ie/Pdf/ServiceOriented1.pdf. ISBN: 0-7695-0915-0; DOI: 10.1109/APSEC.2000.896702.

[18] **H. Brunner,** “Service-Based Architecture,” *Best's Review,* Vol. 104, no. 8, pp. 96-96, December 2003. 1527-5914.

[19] **M. Mullender and M. Burner,** "Application Conceptual View," *Microsoft Developers Network Library.* [Online] July 2002. [Cited: October 24, 2006.] http://msdn2.microsoft.com/en-us/library/ms977997.aspx.

[20] **M. Jeckle and E. Wilde,** "Identical Principles, Higher Layers: Modeling Web Services as Protocol Stack," *XML Europe 2004.*Amsterdam : s.n., 2004. Available at http://dret.net/netdret/publications#wil04g.

[21] **United Nations Economic Commission for Europe,** "United Nations Directories for Electronic Data Interchange for Administration, Commerce and Transport ," *United Nations Economic Commission for Europe.* [Online] United Nations Economic Commission for Europe. [Cited: March 31, 2007.] http://www.unece.org/trade/untdid/welcome.htm.

[22] **G. Kiczales, et al.** "Aspect-Oriented Programming," *Proceedings European Conference on Object-Oriented Programming.* Vol. 1241.s.l. : Springer-Verlag, 1997, pp. 220-242.

11 Resümee

Edasiühilduv veebiteenuste esituskiht

Magistritöö

Siim Karus

Töös uuritakse, kuidas luua edasiühilduvaid esituskihte veebiteenustele. Töö eesmärk on luua juhised, mida järgides on võimalik luua edasiühilduvaid esituskihte. Töös kasutatakse neid juhiseid, et luua uus esituskiht ning vaadeldakse uue esituskihi kasutamiselevõtuga seotud probleeme ning kasutegureid.

Esimestes peatükkides kirjeldatakse edasiühilduvuse nõudeid ning vaadeldakse erinevaid muutusi tarkvaras. Seejärel vaadeldakse esituskihi komponentide tüüpe.

Järgnevalt uuritakse levinud komponentide edasiühilduvust. Vaatluse all on nii juhtelemendi taseme lahendused kui ka konteineri taseme lahendused. Tuuakse välja seosed teenusbaseeruva tarkvaraarhitektuuri ja edasiühilduva arhitektuuri vahel.

Kasutades olemasolevate vahendite analüüsist saadud kogemusi kirjeldatakse nõuded edasiühilduva esituskihi loomiseks ning pakutakse lahendusi probleemidele, millega vaadeldud vahendid ei tegele. Lahendusi pakutakse kahele peamisele probleemile: tulevikukindlate ja edasiühilduvate suhtlusprotokollide valik ning viisakas reageerimine ootamatustele esituskihis.

Leitud lahenduste ja vajaduste põhjal tuuakse välja juhised edasiühilduva esituskihi loomiseks. Uuritakse nende juhiste rakendamise automaatse kontrolli või rakendamise võimalikkust ning tuuakse paralleele tooteliini arengu juhistega.

Väljapakutud juhiseid kasutatakse veebiteenusele uue esituskihi loomisel ning vaadeldakse esituskihi uuendamise tulemusi.

Uurimuse tulemuseks on juhised edasiühilduvate kasutajaliideste loomiseks. Juhiste hindamise eesmärgil on neid järgides loodud uus esituskiht olemasolevale veebiteenusele.

12 Appendix

I. Relations between new software requirements and software evolution

		REQUIREMENTS					
		new product family	new product	improvement of functionality	extend standard support	new version of infrastructure	improved quality attribute
CATEGORY OF EVOLUTION	**split of software product line**	🟥					
	derivation of product line architecture	🟥					
	new component	🟥	🟦				🟩
	changed component		🟦				🟩
	replaced component		🟦				🟩
	split of component						🟩
	new relation between components						
	changed relation between components						
	new framework implementation			🟥			
	changed framework implementation				🟥		🟩
	decreased functionality in framework implementation					🟥	
	increased framework functionality			🟩			
	solving in external component						

Legend	🟥 Always major impact	🟦 Sometimes major impact	🟩 Minor impact

Table 1. Relations between new software requirements and software evolution

II. Checkbox tree supporting JavaScript code

```
// Checks all inputs with given name
function setInputCheck(name, check)
{
   inputs = document.getElementsByName(name);
   for (i = 0; i < inputs.length; i++)
   {
      inputs.item(i).checked = check;
   }
}
// Finds the next element el-s sibling element with name name
function findNextElementByTagName(el, name)
{
   if(el == null)
   {
      return null;
   }
   var a = el.nextSibling;
   while(a != null)
   {
      if(a.nodeType == 1 && a.tagName.toLowerCase() == name)
      {
         return a;
      }
      a = a.nextSibling;
   }
   return null;
}
// Finds the previous element el-s sibling element with name name
function findPrevElementByTagName(el, name)
{
   if(el == null)
   {
      return null;
   }
   var a = el.previousSibling;
   while(a != null)
   {
      if(a.nodeType == 1 && a.tagName.toLowerCase() == name)
      {
         return a;
      }
      a = a.previousSibling;
   }
   return null;
}
// Cheanges subtree visibility
function changeSubTreeVisibility(image)
{
   var subtree = findNextElementByTagName(image, "div");
   if(subtree == null)
   {
      return;
   }
   if(subtree.className == "subtree")
   {
      subtree.className = "subtreehidden";
      if(image.nodeName.toLowerCase() == "img")
      {
         image.src = "graafika/silk013/icons/bullet_toggle_plus.png";
         image.alt = "Näita alampuud";
      }
   }
   else if(subtree.className == "subtreehidden")
   {
      subtree.className = "subtree";
      if(image.nodeName.toLowerCase() == "img")
      {
         image.src = "graafika/silk013/icons/bullet_toggle_minus.png";
         image.alt = "Peida alampuu";
      }
   }
   else if(image.nodeName.toLowerCase() == "img")
   {
      image.src = "graafika/silk013/icons/bullet_white.png";
   }
}
// hides subtrees that have no nodes checked
function hideSubtrees()
{
   var divs = document.getElementsByTagName("div");
   for(i = 0; i < divs.length; i++)
   {
       if(divs.item(i).className == "tree")
       {
          hideSubtree(divs.item(i).firstChild);
      }
   }
}
// hides of el subtrees that have no subtrees checked
function hideSubtree(el)
{
   var a = el.firstChild;
   var img = null;
   var bHide = true;
   while(a != null)
   {
      if(a.nodeType == 1)
      {
         if(a.tagName.toLowerCase() == "img")
         {
            img = a;
         }
         else if(a.tagName.toLowerCase() == "input" && a.type == "checkbox" &&
a.checked)
         {
            bHide = false;
         }
         else if(a.tagName.toLowerCase() == "div" && a.className == "subtree")
         {
            if(!hideSubtree(a))
            {
               bHide = false;
            }
            else if(img != null)
            {
               changeSubTreeVisibility(img);
            }
            img = null;
         }
      }
      a = a.nextSibling;
   }
   return bHide;
}
// called when page is loaded
function onLoad()
{
   hideSubtrees();
}
```
Listing 6. JavaScript code supporting chekbox tree sample.

III. Glossary

Service An abstract resource that represents a capability of performing tasks that form a coherent functionality from the point of view of providers' entities and requesters' entities.	Teenus Abstraktne ressurss võimega täita pakkujate ja nõudjate jaoks funktsionaalse terviku moodustavaid ülesandeid.
Web service Software service made available over the World Wide Web.	Veebiteenus Ülemaailmse võrgu vahendusel kasutatav tarkvaraline teenus.
Forward compatibility An applications preparedness to changing environment and adoption of technologies.	Edasiühilduvus Rakenduse valmidus keskkonna muutus-teks ja uute tehnoloogiate kasutusele-võtuks.
Extensibility An applications capability to be extended.	Laiendatavus Rakenduse laiendamise võimalikkus.
Abstraction Withdrawal or separation of components.	Abstraktsioon, üldistus Osiste eraldamine või mitte arvestamine.
n-tier application An application that can be functionally separated into *n* abstract layers.	*n*-kihiline rakendus Rakendus, mida saab jaotada *n* abstraktseks funktsionaalseks kihiks.
Componentization Separation of an application into separate executable components.	Komponentideks jaotamine Rakenduse eraldiseisvateks täidetavateks komponentideks jaotamine.

Future proof design Solutions design where the solution can be used with future technologies without having to make any changes to the solution itself.	Tulevikukindel arhitektuur Arhitektuur, mida saab muutmata kujul kasutada uute tehnoloogiatega.
Web Services The programmatic interfaces used for application to application communi-cations over the World Wide Web.	Web Services Programsed liidesed, mida kasutatakse rakendustevahelisel suhtlemisel ülemaa-ilmses võrgus.
Control The complete apparatus used to control a mechanism or machine in operation.	Juhtelement Seade mehhanismi või masina töö juhtimiseks.
Container Any object that can be used to hold things.	Konteiner Mistahes objekt, mida saab kasutada asjade hoidmiseks.
Widget A combination of a graphic symbol and some program code to perform a specific function.	Vidin Graafilise sümboli ja programmikoodi kombinatsioon kindla funktsiooni teostamiseks.
Template A gauge, pattern, or mold, commonly a thin plate or board, used as a guide to the form of the work to be executed.	Mall Näidik, muster või valuvorm, mis esitab täitmisele võetava töö struktuuri.
Graceful degradation Easy, elegant lowering of the rank.	Sujuv mandumine (pehme degradeerumine) Lihtne ja elegantne tähtsuse vähendamine.