Exercise 3: Software Architecture in Practice Case: Beer Web Store

Anders H Poder, Lars Kringelbach

Department of Computer Science, University of Aarhus
Aabogade 34, 8200 Århus N, Denmark

Group 11 - Kilo
19951439, 20074842
{ahp, u074842}@daimi.au.dk

2008-03-17

1 Software Architecture – Beer Web Store

A new Beer Web Store for speciality beers has the following main use case:

> The customer browses the beer catalogue for descriptions and reviews. The customer selects appropriate beers for purchase and these are added to his shopping basket. When the customer checks out, payment is validated and he is presented with a receipt

As a first step, the software architect of this systems has decided to use a layered architectural style (with clients, server, and a database in separate layers). He has decided that the communication to the database will be done via a Façade pattern. The system will be built in Java.

1.1 Software Architecture - Question 1

> *Outline how you would approach the tasks of creating the architecture for the Beer Web Store. Consider, e.g., which steps would you takes and in which order?*

Naturally there are many ways to create an architecture, yet as the focus is on the architecture then ADD (Architecture Driven Design) is an obvious choice. ADD is merely a method that employs many of the techniques related to Software Architecture in a predefined structure.

1.1.1 Choose the module to decompose

As this is the beginning of the architectural design then the module is the entire system. In later iterations the sub-components (the layers, the components in the layers, etc.) will be decomposed.

1.1.2 Refine the module according to the following steps

a. Choose the architectural drivers Here the quality attributes of the system would be defined by the stakeholders of the system and then a vote would be used to determine which of the quality attributes are architectural drivers. The quality attributes would be expressed as quality attribute scenarios.

b. Choose an architectural pattern Based on the architectural drivers selected in *a*, use the mapping in [Bass et al., 2003] as well as other literature and experience to determine some tactics for achieving the architectural drivers for the system. Then select some architectural patterns to realize the tactics.

c. Instantiate modules using multiple views Based on the tactics and patterns found above the architecture for the system may be expressed using UML. It is *always* required to use multiple views in order to properly describe the architecture. A good approach is the ”3 + 1” structure, which uses three viewpoints; Module viewpoint, Component & Connector viewpoint and Allocation viewpoint. These three viewpoints are expressed in UML using Package diagrams, Object and sequence diagrams, and deployment diagrams respectively. These diagrams should in the first iteration only look at relations and interactions between the main components (the layers).

d. Define interfaces Define the interfaces between the main components (the layers).

e. Verify scenarios and use cases Make sure that the steps a - d covered all the parts of the scenarios. If not, include the missing parts and updates the scenarios/architecture if inconsistencies are found.

1.1.3 Repeat the steps above for every module that needs further decomposition

The first iteration only look at the overall architecture (the location and communication between layers). The following iterations will decompose the individual layers and as needed the individual components in the layers.

1.2 Software Architecture - Question 2

Give concrete examples of what elements, relations, and structures as defined in [Bass et al., 2003] could be in relation to an architecture for the Beer Web Store.

1.2.1 Elements

Based on the present decomposition the elements that are known are the layers, where each layer is an element as described below.

Clients The clients access the server in order to perform some action.

Server The server is accessed by the clients and when needed access the database via its Façade pattern.

Database The database exposes its interface to the server via a Façade pattern.

The above is quickly sketched in the box-and-line drawing below in figure 1, where the layering may also be seen.

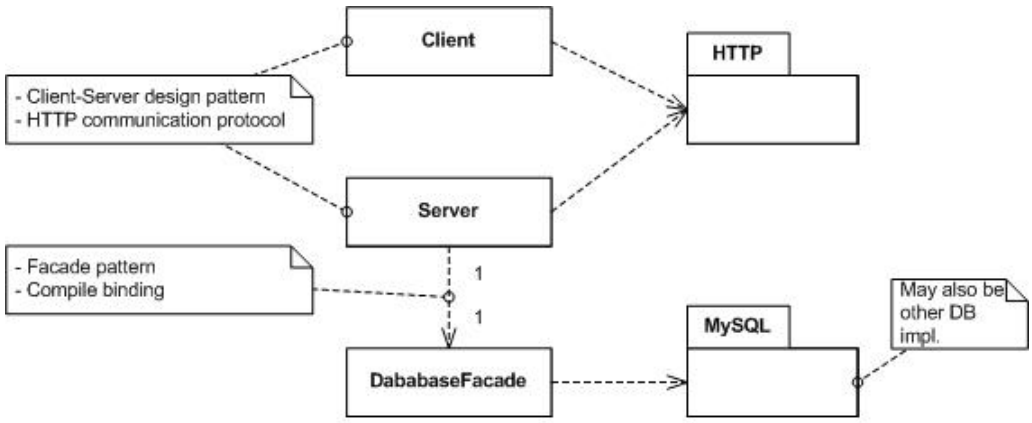


Figure 1: Element overview

1.2.2 Relations

Figure 1 shows some of the relations, namely the statical, yet it is also important to know the dynamical behavior. These may be expressed by e.g. a sequence diagram, as shown in Figure 2. In this it may be seen that the Client and Server has a weak semantic relation through the http protocol, and the Server and Database has a strong syntactic relation from a compiled compatibility.

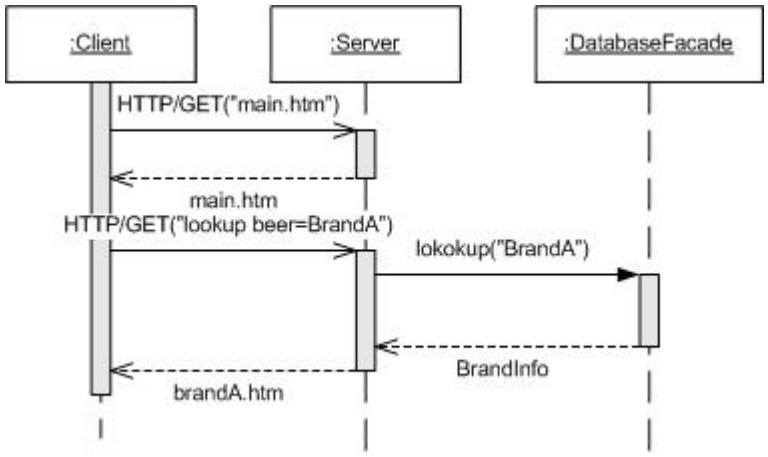


Figure 2: Relation overview

1.2.3 Structures

There are many structures which may be used to model a software system. Common for all of them is that they present a certain view of the system. Very rarely is a single structure sufficient to express the architecture of a system. For this reason many structures are combined, each structure being represented by one or more views, to create a sufficient representation of the software system.

The most popular combination of structures are the Module, the Component-and-Connector and the Allocation. Each of these three structures focus on a specific aspect of the software architecture; the static elements and their relation, the elements dynamic relations and the elements deployment on hardware, respectively.

In each structure there is a number of views. Which view, and indeed which structure, that apply is quite dependent on not only the system, but also the iteration of the architectual analysis. In the first iteration of the beer-store it would make sense to choose more overall views, e.g. the Module Layered or Module Decomposition, whereas e.g. the Module Class would be too fine-grained for the first iteration. For the C&C it would make sense to start with the Client-Server, and finally for the Allocation the Deployment view may give a good indication of the physical location and requirements of the different elements.

1.3 Software Architecture - Question 3

Apply [Perry and Wolf, 1992]'s model of software architecture as

$$\textit{Software Architecture} = \{\textit{Elements, Form, Rationale}\}$$

to the Beer Web System

Perry and Wolf suggest separating the model of the architecture into three components, as mentioned in the question. The components relate in the sense that the *elements* has a particular *form* and the elements and their relations are chosen based on a *rational*.

There are three classes of architectural elements defined.

Processing elements Transforms data elements.

Data elements Information to be transformed or used.

Connecting elements The *glue* that hold the architecture together. May at times be *Data elements*, *Connecting elements* or both. The connecting elements are the relations between the different elements, e.g. procedure calls or message parsing.

The architectural form describe the architectural properties and the relationships and their relative importance. This is important as some architectural properties may be vital to the design and others may be gold-plating. These properties are used to put constraints on the elements and also define the relative importance of these constraints. The relationships are similar to the properties except they are used to put constraints and relative priority on the relations between elements.

A software architecture is a selection and description of architectural choices. These choices are, however, difficult to convey without including the reason for the choices made, for the elements and form not chosen and for choices not made due to lack of information or because they were deemed irrelevant. This information is kept in the Rational.

When the above is applied to the information present for the Beer Web Store three processing elements, one data element and two connecting elements may be defined.

1.3.1 Elements in Beer Web Store

- Processing elements.

Client Receive user entries and process them into http requests

Server Receive http requests and transform them into high level database lookup

DatabaseFacade Transform high level database lookup into proprietary database lookup

- Data elements.

Request data The packets of data flowing from the client to the server and form the server to the database, e.g. Beer type, Credit card information, search refinement parameter, etc.

- Connecting elements.

HTTP Protocol for connecting the server and client (loose coupling).

Database façade methods Protocol for connecting the server to the database (medium-loose coupling).

1.3.2 Form in Beer Web Store

Properties The Server and the database, hidden behind the Façade, is the backbone of the Beer Web Store. If either is unavailable then the other is irrelevant. Multiple instances of both Server and Database may be running simultaneously in order to supply hot-swapping. The importance of up-time (availability), security (payment), usability (transmission abilities), etc. must be defined here.

Relationship The Client may only see the server through its HTTP interface, and the Server may only access the database through the Database Façade. Other demands to the relationship (certain commands to the database may only be accessed when using https).

1.3.3 Rational in Beer Web Store

Finally it is important to describe the decisions made when defining the form - why is availability important? Why is it OK to sacrifice availability for some (security) functionality?

1.4 Software Architecture - Question 4

Reflect on what happens if the words "software" and "computing" are removed from the definition of 'software architecture' in [Bass et al., 2003]

If the references to software and computing is removed the definition talks about elements, structures and properties, and their relations. This definition may be used about anything which may be broken down into generalised components. As an example could be used the construction of a rifle. Describing the elements that goes into making a rifle (hammer, barrel, ...), the structures that comes from putting them together, and the individual relationships that must be met (form fit, etc.). The elements may fit together in different ways and multiple structures may be required to complete the description. (the hammer up, the hammer down, etc.). The properties are in fact only the externally visible to the given element, as those are the only properties that are relevant to the architecture. These details include things like the strength of the metal, which may be different for the hammer and the pipe, the price range, production time, etc. while things like the cheek-pad may be optional.

The description of an architecture, as defined by [Bass et al., 2003] is therefore very general and apart from the use of words like "software" and "computer", may apply to any architecture, be it a building, a production line for a rifle, or a embedded software system.

1.5 Software Architecture - Question 5

The architect decides to create a full architecture description before embarking on any implementation of the system. Discuss pros and cons of taking that approach

There are generally two approaches to creating an architecture, as defined by Bass et. al. and that is to define it in advance, and only define the overall structure and then let it grow from the following implementations.

There are also two reasons for creating an architecture; in order to create a product that exhibits some of the quality attributes defined in the architecture (modifiability, usability, ...) and in order to reuse the architecture (or part of the architecture) as a product line for creating other products. This latter reason is described in section 5.2.

Creating an architecture up front allows for the definition of all properties of the architecture; the quality attributes and their individual priorities, the individual elements/components and how they interconnect both at runtime and statically, etc. This has the benefit that when the design and implementation phase is begun it may easily be broken down into parallel activities and integrated later, as everyone has a clear picture of what is important and what the interfaces are.

Unfortunately defining the architecture fully up front only makes sense if there is enough information in advance to create the architecture from. Very often there are "dark horses"; things that do not become apparent until the design and implementation phase. If it is not believed that sufficient information exist to fully define the architecture, then doing so may not only be a waste, it may lead to the very dangerous loose-loose principle, where e.g. an overhead is introduced to service modifiability, only to be hacked in order to service a performance issue that was not apparent at the architectural design phase. This means

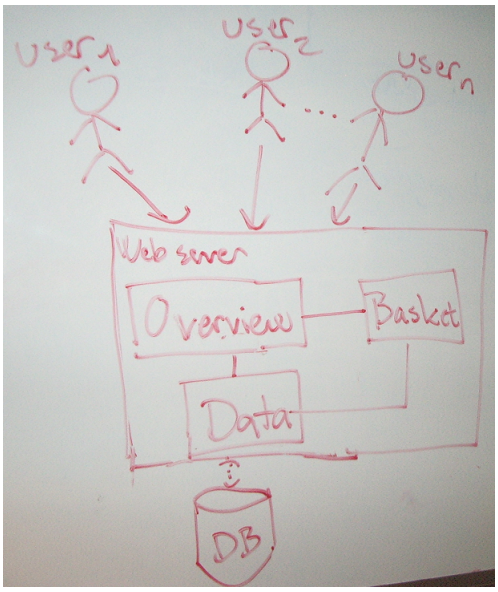
that part of the overhead is preserved (in memory-footprint if nowhere else), without preserving any of the modifiability advantages.

Another requirement to defining the architecture fully up front has to do with the organization. It is a waste to create a fully defined architecture with focus on parallel implementation, if there is only one in-house developer to implement the product.

Assuming that the architect in question has followed the course Software Architecture in Practice, he has obviously looked at the architectural properties, the stake holders, the organization and the requirement and deemed that there is sufficient grounds to fully specify the architecture in advance, with the risks and possible gains that it has (parallel implementation, out-sourcing of the design/implementation, etc.).

2 Architectural Description – Beer Web Store

The architect for a new Beer Web Store for speciality beers has drawn the following figure of the architecture together with his customers. The system is to support web users that may purchase beers using a web



browser. The system runs on top of a database (“DB”), contains domain model (“Data”), an overview page (“Overview”), and a “Basket” subpart.

2.1 Architectural Description - Question 1

> *Which structures does a system such as the above exhibit? Give examples of elements and relations pertaining to each structure*

The information depicted in the figure above shows several software structures of the Beer Web Store to some extent.

2.1.1 Uses

The *uses structure* shows how elements are related with regard to usage. One perspective of the uses structure is that the elements in the figure are “Overview”, “Basket” and “Data”, which are all related and the database, which is only related to “Data”. This shows that the overview page and the basket has been decoupled from the database, which therefore can be replaced without affecting them.

2.1.2 Layered

Another perspective is that the users, the web server and the database, are elements where the relations show that the users use the web server and the web server uses the database. This also shows the that the architecture is a *strictly layered structure* where the users cannot access the database directly.

2.1.3 Client-Server

As there are several users, the elements of the *strictly layered structure* can also be seen as a *client-server structure*, where the users are clients and the web server is a server. The web server can also be seen as a client to the database server. This would make it possible to achieve better performance if several web servers are used to make load balancing of the user requests.

2.1.4 Deployment

With regard to a *deployment structure*, the figure shows that the overview page, the basket and the domain model (elements) are allocated to (relations) on the web server and the database is allocated to its own server.

2.2 Architectural Description - Question 2

> *Which viewpoints are relevant when describing the example? Give examples of partials views for each viewpoint*

Selecting the relevant views shall actually be performed on the basis of the architectural drivers which have not been determined for the Beer Web Store. There are however several models that provide a number of views that are often useful, e.g. the "4+1" model [Kruchten, 1995] and the "3+1" model [Christensen et al., 2004] and many others.

As described in section 2.1 the figure shows parts of a module viewpoint (uses, layered), component & connector viewpoint (client-server) and an allocation viewpoint (deployment) which are the recommended viewpoints in [Christensen et al., 2004].

These viewpoints are all relevant to the Beer Web Store example. The Module viewpoint describes the internals of the web server and how the functionality is decomposed into units. The Component & Connector viewpoint describes the runtime relations between the units and the Allocation viewpoint describes the environment of the running system.

In the following sections we will give examples of views for each of the three viewpoints.

2.2.1 Module Viewpoint

Figure 3 shows a package diagram of the Beer Web Store and a decomposition of the web package. The Web package contains the elements used for the web interface, the Overview page, the basket and the domain model. The domain model depends on a Database package that contains the database façade and the database façade uses some database package library, e.g. MySQL.

2.2.2 Component & Connector Viewpoint

There are only two active components in the Beer Web Store, a web server, e.g. Apache, and a database server. An active objects diagram will not be included here.

Figure 4 shows a sequence diagram where a user searches for a an item "tuborg" and adds one of the search results to his basket.

The domain model accesses the database through the database façade when needed. This can be expressed in another sequence diagram. Sequence diagrams showing the checkout including payment must also be made.

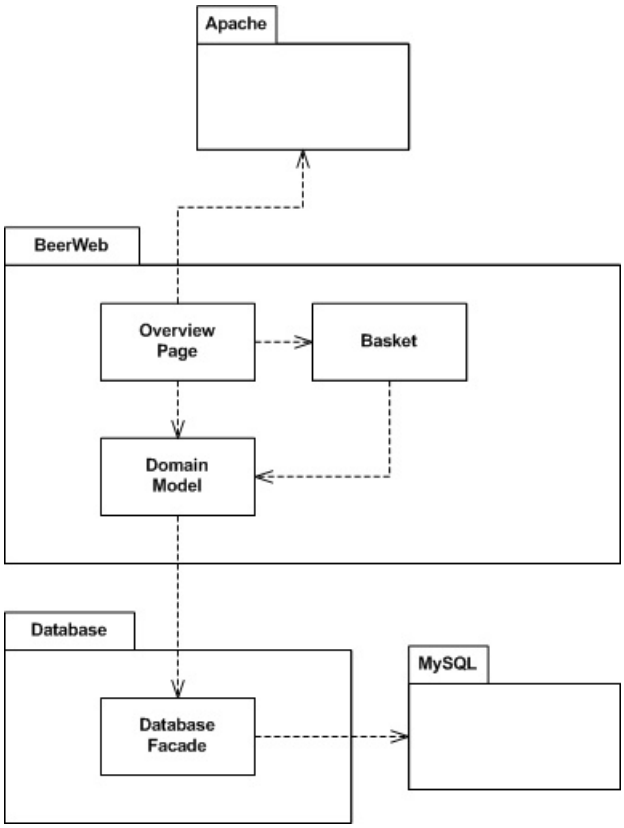


Figure 3: Package diagram for the Beer Web Store

2.2.3 Allocation Viewpoint

Figure 5 shows a deployment view of the Beer Web Store where the web server and the database is located on two different servers. Clients can connect to the Beer Web Server using a http web browser. The web server and database server are connected through a local area network.

The application code (Beer Web) and the database façade are both deployed on the web server.

2.3 Architectural Description - Question 3

Argue for benefits and liabilities of describing software architecture via a box-and-line drawing such as the above

The benefits of a box-and-line drawing is that it can be written and ”understood” by almost anyone, and in the initial phases of the architectural design they may be fine. They are fast to draw and can be shown to non-technical stakeholders. It is kind of like simple sign-language - everyone understands nodding and shaking ones head or indicating a direction as long as it is for general information.

Unfortunately the general information is where the advantages stop. For simple sign-language, it may quickly come to an argument if one wishes to know if the indication of ”this way” means ”a short way that way” or ”a long way that way”, and the same is true for box-and-line drawings. ”How many

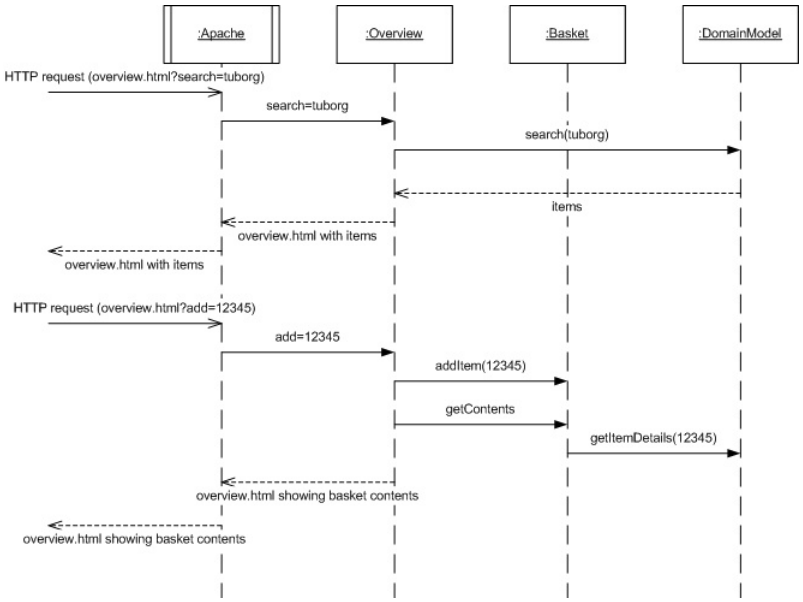


Figure 4: Sequence diagram showing a search and adding of an item

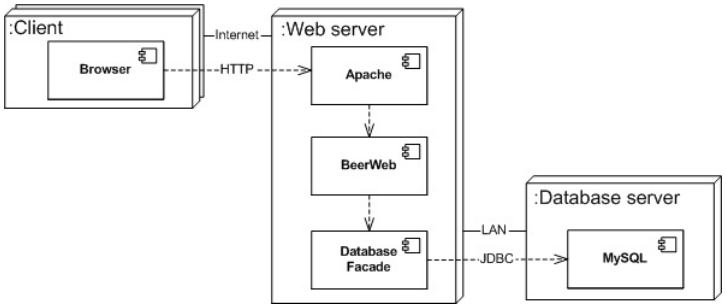


Figure 5: Deployment view of the Beer Web Store

clients does the system support?". "How strong is the binding indicated by the line - do all lines mean the same?" There is merely a line, so what does that line mean?

Working as a developer/tester/engineer means that everything must be quantifiable in some way, and box-and-line drawings do not offer this unambiguous quantifiability. For this a clearly defined modelling language, without ambiguity as to the meaning of the model, is needed. An example of such a language is Unified Modelling Language (UML), which would allow the box-and-line drawing to be represented as one or more diagrams solving all the ambiguity of the box-and-line drawing with only a single requirement; one must know the language, which is indeed one of the reasons while box-and-line drawings are still widely used - there is no single unambiguous modelling language known by all stakeholders, and when that is the case it may be more confusing then beneficial to spend time making sure the diagram is clear and unambiguous, if that clarity is derived from symbolic notation that the readers (stakeholders) do not understand.

2.4 Architectural Description - Question 4

Discuss what architectural description would be needed if the system was to be realized as a Service-Oriented Architecture

In a sense there are already aspects of SOA in the architecture, as the validation of the credit card information is most likely performed using a service on the internet - not that web services and SOA is equivalent, but they do complement each other nicely.

As SOA consists of a collection of (preferably stateless) well-defined services what needs to be described is, which existing (external) services are needed and which new services has to be created. This may be done using the MacKenzie Topic maps, which describe the static dependencies between the services.

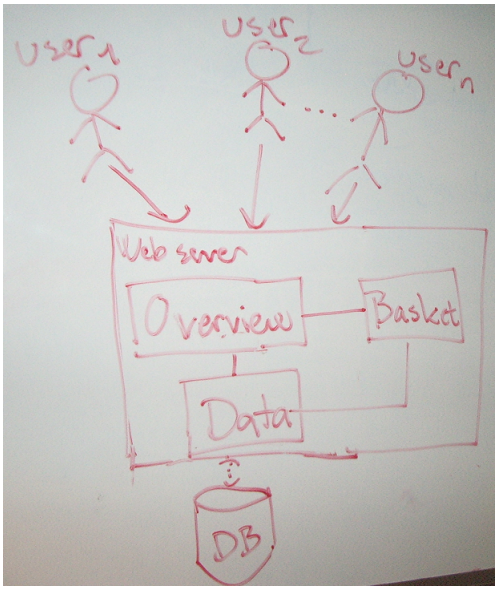
The static description is naturally not sufficient, and two other views may be presented to show the Dynamic interrelations for the services that are provided; Visibility and Interaction. The visibility view describes the awareness, willingness and reachability of a service and the interaction view describes how the service can be used (behaviour model) and the structure and semantics of the information that is provided (information model).

3 Quality Attributes – Beer Web Store

Consider the following main use case for a Beer Web Store for specialty beers:

> The customer browses the beer catalogue for descriptions and reviews. The customer selects appropriate beers for purchase and these are added to his shopping basket. When the customer checks out, payment is validated through the payment service

The main architectural drivers are considered to be performance and availability. Moreover, the software architect for the system has created the following box-and-line drawing documenting his initial ideas of the system:



3.1 Quality Attributes - Question 1

> *Describe technique(s) for architectural requirements capture that are applicable to the above case*

Defining requirements for an architecture is not like defining the functional requirements; actually many software professionals believe that they are indeed orthogonal. This is supported by the fact that a set of functional requirements may be implemented by a number of different architectures.

Finding out the requirements for the architecture is therefore more a matter of finding out what is important, more than it is to define what the product must do. The people who are able to define what is important to the architecture is naturally the stakeholders that the architecture affects, i.e. the owner, the user, the developer, the special interest groups, etc. Getting all of these peoples, often conflicting, requirements assembled and, even more importantly, prioritised, can be a difficult task. One way of doing so is to define scenarios, like the one mentioned above, and then having the stakeholders vote at the importance of the scenarios.

Naturally in order for the scenarios to be able to be used to clarify the architecture it must be possible to quantify them, and place them in different categories. This requires that the above scenario(s) are extended with e.g. response time, how many times the user should be able to perform an operation before experiencing a fault (down-time).

Once the scenarios are clarified and voted on, then the top priority scenarios may be formalized and it may be determined which quality attribute(s) they involve. For examples lets assume that the above scenario was selected as top priority.

As it may be seen, the scenario(s) focus primarily on *Availability* (how rare should it be that a user experience non-responsiveness and how many simultaneous users can the system support), *Usability* (how easy should it be for the user to cancel a transaction halfway through and how easy should it be to pick up a previous transaction), *Performance* (how big a delay can the user expect when performing an operation like a lookup in the database), *Security* (how to avoid someone using stolen credit cards and also prevent a ”hacker” from stealing the credit card information either during the transaction or from the Beer Web Store’s database.

The scenario does not include all aspects of the architecture, though. Another scenario may deal with how long it may take to alter a product built on the architecture (*Modifiability*) - this may be particularly interesting to the owner. Another quality that has not been dealt with in the mentioned scenario(s), is how easy it is to check a product developed based on the architecture for faults (*Testability*), which is of great interest to the owner and the developers.

Naturally there are many more scenario required in order to properly cover all aspects of the quality attributes.

3.2 Quality Attributes - Question 2

Give feasible architectural requirements for availability and performance for the Beer Web Store using such techniques (at least one for each quality attribute)

The following scenarios are a possible outcome of prioritization and clarification of quality attribute scenarios as described above:

Scenario(s):		The customer browses the beer catalogue for descriptions and reviews and receives a timely response
Relevant Quality Attributes:		Availability, (Performance)
Scenario Parts	**Source:**	The user
	Stimulus:	Browsing the beer catalogue
	Artifact	The system
	Environment:	During normal operation
	Response:	Web page presented to user
	Response Measure:	The result is presented within 5 seconds 99% of the time, 30 seconds 0,9% of the time and more than 30 seconds 0,1% of the time
Questions:		
Issues:		The response measure is also dependent on the speed of the users internet connection and computer hardware.

Scenario(s):		When the customer checks out, payment is validated through the payment service within 30 seconds.
Relevant Quality Attributes:		Performance, (Availability)
Scenario Parts	**Source:**	The user
	Stimulus:	The customer checks out
	Artifact	The system
	Environment:	During normal operation
	Response:	Result of payment validation if presented to the user
	Response Measure:	Payment is validated through the payment service within 30 seconds
Questions:		
Issues:		The response measure is also dependent on the performance of the payment service.

In a real life system there will of course be many more scenarios to describe the architectural requirements.

3.3 Quality Attributes - Question 3

Argue how tactics and/or styles may be used to resolve the requirements

As we have only defined the Availability and Performance requirements, only these will be dealt with here, yet naturally requirements concerning the other quality attributes must also be handled.

3.3.1 Availability

Availability has two main concerns; finding out that a resource is unavailable and if it is, then restart or switch to another resource.

Finding out if a resource is down may be done using one of the fault detection tactics; ping/echo, heartbeat or exception. Due to the loose coupling between the client (browser) and the server, the exception solution is not beneficial. Since it may be assumed that the supervising process (remotely deployed) (the process monitoring if the server is down) is stable, then a heartbeat is sufficient to ensure availability, meaning that the web-server will issue a heart-beat every e.g. 1 second to a maintenance process, and if one (or perhaps more) heartbeats is missed then the maintenance service can act accordingly.

The proper response to a faulty server may be many things, as defined by the tactics group *fault recovery*. As the requirement for availability stipulated a three stage fault recovery, the server or servers (as it may be required to have multiple serves to do load-balancing, depending on how many users must be supported) can service the 5 second requirement, and then if an error occur (as detected by the heartbeat), a restart or swap to another server/database can be performed in 30 seconds. As 30 seconds is insufficient to boot a web server and a database, the *Spare* tactic would be insufficient. At the same time having *Active redundancy* would be over-kill, as this kind of immediate switch-over is not required. For these reasons *Passive Redundancy* is sufficient.

As it is an Internet system, it will also be sufficient to simply restart the session, and no *Shadow*, *State resynchronization* or *Rollback* is needed in the main web server. The user may experience being thrown

back to the main page, yet this is deemed acceptable, under availability and performance, yet it may not be in Usability. During a payment transaction, however, it must be ensured that the purchase is either completed or not-completed - halfway completed is not acceptable. For this it will be beneficial to use the *Rollback* tactic on the *Basket* component.

Finally a web-server may be deployed multiple times on the same machine, allowing for the recovery or prevention of the lesser faults, by simply re-spawning the process that has become non-responsive, utilizing the *Process Monitor* tactic. This is e.g. done by the Apache-server.

As it is not only the web server which may become non-responsive, it is also required to monitor the database. However, as a non-responsive data-base may be discovered by a non-responsive server, it is possible to use the *ping-echo* tactic with the database, and only perform the ping-echo if the server becomes non-responsive, to find out how much must be restarted (only the server or both the server and the database). Depending on the binding between the database and the server, it may also be possible to use the exceptions tactic between the server and the database, so the server itself can restart the database, if an exception occur in a call to the database.

3.3.2 Performance

As we have all experience the irritation of a slow web-site, and as the competition on the internet is growing, it is important to achieve propper performance.

Due to the decreasing price of hardware and the increasing price of development time, it is often cheapest and certainly simpler, to use the tactic *Increase available resource*; introduce load balancing with multiple parallel servers thereby increase performance during heavy-load situations, ensure that the serves have sufficient network bandwidth, RAM and CPU strength to handle many simultanious database look-ups - here the tactic of *Maintain multiple copies of data or computations* may be used to speed up database-look-up by caching the most popular searches, or cache the entire result in preparation for a search refinement. If the database is remotely located the caching of results becomes even more important.

There are many other tactics which may be employed, yet given the nature to the application, it is not believed that the biggest resource consumers are under development control, for building the web-server from scratch to support better concurrency or using non-standard performance tweaking algorithms for the database-lookup, will most likely become inconceivably expensive compare to simply increasing the computational resources.

3.4 Quality Attributes - Question 4

Discuss where Service-Oriented Architecture can play a role with respect to the Beer Web Store? Consider the quality implications

As SOA is all about standardised interfaces supplying a functionality which may be used by many different products, the areas where SOA makes sense is naturally between the components.

Instead of creating a web-server which access a database through a specific façade, one may argue whether the desired functionality may not be generalised. ”Look-up in database”. This may easily be hidden behind a standardised service, e.g. a web-service, meaning that anyone on the WWW who supplies a database service may be used to host the beer database, and if one becomes dissatisfied with the database provider, it is transparent to simply switch to another.

Taking this one step further, perhaps the database service may be extended with a data-base type, meaning that the Beer Web Store’s domain model is exposed in a service, which would allow for not only

displaying ones own brands of beer, but through a discovery service, also beer-brands by other brewers on the net who implement this service. This will also point the way for a more standardised look-and-feel when shopping for beer.

Naturally just because a web-service (or other service) is used it does not mean that it has to be public, jet it does have some advantages.

Finally the WS-Security web-service may be used to handle payment, and also possibly, if the database or Beer Web Store Domain model is exposed through a web-service, it may also be beneficial that not everyone may have the ability to update the domain model or database, so here the WS-Security service may also come in handy.

Naturally this generality do not necessarily come free. A look-up in a database behind a service, especially a web-service, is far slower than a direct look-up in a local database, thereby influencing Performance poorly. On the other hand the use of web-services may increase Availability, as the functionality is deployed on multiple servers, thereby reducing the chance of the system becoming fully non-responsive. Naturally the critical path must be evaluated, as if a link on the critical path chain become non-responsive then the entire system becomes non-responsive.

3.5 Quality Attributes - Question 5

Reflect upon what the role of quality attributes in software architecture is

The quality attributes are vital to the software architecture, as they not only define the requirements to the software architecture (along with the scenarios), but also describe their relative importance. The quality attributes therefore allow the architecture to be evaluated in terms of how well it supports or enables a system's desired external (e.g. availability and performance) and internal behaviour (e.g. testability and modifiability). [Sarjoughian, 2002]

4 Architectural Design – Beer Web Store

Consider the following main use case for a Beer Web Store for specialty beers:

> The customer browses the beer catalogue for descriptions and reviews. The customer selects appropriate beers for purchase and these are added to his shopping basket. When the customer checks out, payment is validated through the payment service

The main architectural drivers are considered to be performance and availability. The software architect and stakeholders of the system have based on this outlined the following two central Quality Attribute Scenarios:

- A customer makes a payment to the system in normal mode. The payment is processed within 10 seconds
- The payment service fails during normal operation. The system detects this. The administrator is notified and the system continues in degraded mode until the payment service is made available

4.1 Architectural Design - Question 1

> *Give examples of performance tactics respectively availability tactics that may be applied to the scenario. What are the consequences of applying these tactics?*

This topic has already been breached in Section 3.3, so this section may be seen as a continuation of that.

Assuming that the 10 seconds is an increase in the requirements to the performance, and that the payment service is one that we can actually control (if it is a web-service supplied by a third party (e.g. PBS) then we may have no influence on its performance).

If this is the case then the performance tactics which may be employed here could be *introduce concurrency*, as some of the validations may be done in parallel, e.g. validate the credit card, prepare shipping information, access logistics system. Naturally this will require a roll-back system if it turns out that the payment did not go through. This approach is acceptable as the performance requirement is only valid in the instance where the payment is valid - how long it takes if the user enters invalid information is less important.

As there is no clarification of the response measure, it it could be assumed that the requirement must be met 100% of the time, yet naturally this is not possible. It may however be required that multiple instance of the payment service or validating server are running in order to service multiple purchases simultaneous, as required by the *Availability*.

If the payment is considered of higher priority (*Scheduling Policy* tactic), so that other functionality (e.g. browsing) may be handled slower while the payment is performed, then handling the scheduling of these tasks may also help to meet the 10 second requirement.

Naturally implementing these tactics do not come for free; increasing hardware performance and bandwidth cost money, controlling the scheduling adds complexity and thereby reduce readability and modifiability.

The second scenario focuses mainly on Availability, and actually goes against some of the previos scenarios by stating that the payment service is allowed to be ”down” for a longer period of time, pending an administrator. This approach has one big benefit; it is simple. No hot- or cold-swapping of payment services, and the tactic *Spare* may be employed.

If the failure is part of an attack, it is easier to prevent by simply degrading the system and waiting for an administrator (may increase *Security*).

The second scenario does include requirements to fault detection, yet these are already described in Section 3.3, and the price of fault detection is also increased cost of hardware and readability/modifiability.

4.2 Architectural Design - Question 2

> *The software architect decides to use a layered architectural style for the system. Discuss quality implications of this choice*

Layered styles come in two variants; the strictly layered and the layered. The strictly layered means that the components in a layer may only access other components in the same layer or the layer directly below it. The Layered may access components in the same layer or any layer beneath it.

When a system is Strictly Layered it gives rise to high *Portability* (a form of *Modifiability*), as it is possible to abstract all dependencies of a given layer by simply abstracting the layer directly beneath it, and if anything in a layer changes then only the layer directly above it must be updated.

This also enables simpler *Testability*, as the introduction of test doubles (fake objects, stubs, spys and/or mocks) must only be performed in a single layer; the one directly below the layer under test.

It may however be difficult to adhere to a Strictly Layered approach, as some functionality may be required on all levels, e.g. the Java Collections, and very often vertical ”layers” are introduced to support this all-layer access, which reduces *Modifiability* if the change is in the vertical ”layer”.

4.3 Architectural Design - Question 3

> *Outline major elements of a possible component and connector structure of the Beer Web Store system*

There are four major structures in Component-and-Connector:

- Process or communicating processes.
  - This structure shows the execution path through a system, and is used to give the engineer an understanding about the dynamic behavior of the system. These structures may help engineers increase Performance and Availability.
- Concurrency.
  - This structure is used to see the parallelism in the system, or part of the system, and when a system, or part of a system, is build around this structure it promotes performance.
- Shared data or Repository
  - This is a very good structure to use to illustrate how the data is processed and consumed to/from the database (Server to/from Database Façade). When a system, or part of a system, is build around this structure it promotes performance and data integrity.
- Client-server

- This is a very good structure to see the protocols used between the Clients and the Server, and the messages they communicate. When a system, or part of a system, is build around this structure it promotes separation of concern (i.e. Modifiability) and allows for Availability tactics like Load balancing.

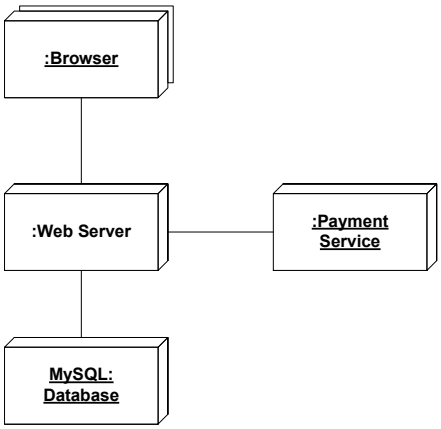
The Beer Web Store has two clear C&C structures, the Shared data structure between the Server and the Database Façade and the Client-server between the Client and Server (see Figure 1).

The Process or Communicating processes is a more fine-grained structure, and an example of this may be seen in Figure 4.

The Concurrency structure is not used in the overall elements, yet may be employed in a given element to achieve more performance.

5 Product Lines – Beer Web Store

Consider a Beer Web Store for buying specialty beers. The software architect has designed a deployment structure of the system as shown below: The main architectural drivers for the system are performance



and modifiability and the following is the main use case for the system:

> The customer browses the beer catalogue for descriptions and reviews. The customer selects appropriate beers for purchase and these are added to his shopping basket. When the customer checks out, payment is validated through the payment service

The company behind the Beer Web Store has decided to build a product line based on the store so that, e.g., sodas or stamps may be sold in a new shop or other delivery platforms such as mobile phones may be used.

5.1 Product Lines - Question 1

How will the choice of a product line approach affect the architecture of the beer web store?

When considering a product line approach for our Beer Web Store, we first need to identify the scope of our product line. In this case the product line is defined to have a narrow scope. The product line supports a web enabled store that sells quantifiable products (i.e. not services). The products may however have different details associated, e.g. beers have a alcohol volume, stamps do not. The web stores must support different user interfaces (but all web based) and on line payment methods.

All these variations and commonalities needs to be identified. The variations needs to be separated so that they either can be created specifically for the actual web store, or so that a common component can be configured to handle the specific product.

The following variation points can be identified from the description above:

Products: The products of the web stores are the main variation point when creating this product line. Some product details will be relevant to all stores, e.g. a product price, and others will be specific to the type of product sold, e.g. alcohol volume as stated above.

User interface: It must be possible to vary the user interface layout with regard to colors and structure of the web page in order to sell the specific product the best possible way. Furthermore the user

interface must be decoupled from the handling of products in order to support several types of web interfaces such as regular browsers and mobile devices.

Payment form: On line payment methods must be supported, but all stores may not want to support all types of credit cards. The supported payment forms must therefore be a variation point in the product line.

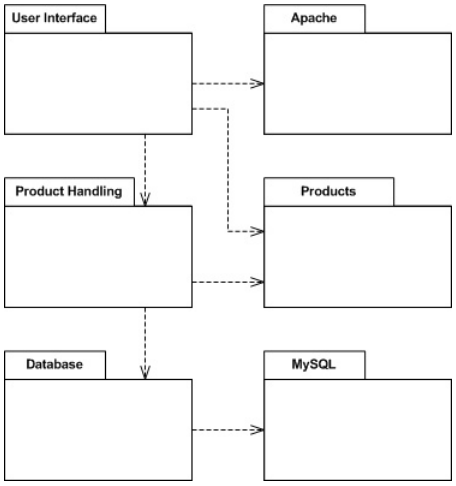


Figure 6: Package diagram for the web store product line

Figure 6 shows an example of a package diagram for a web store product line. It shows that the user interface and the product information has been separated from the product handling and put in separate packages. The user interface may be entirely specific to each web store or the package may contain predefined user interfaces that can be configured e.g. using style sheets. A reactive approach can be used to develop more web interfaces when the need for more features arises. The user interface depends on the product handling package that can handle all kinds of products. The products package can provide an interface that the user interfaces and product handling module can depend on. There need to be some way to define product specific information that can be retrieved from and searched for in the database and presented on the user interface. This concept is not defined any further in this report.

The dependencies to from the user interface and product handling to the products package can e.g. be solved at compile time.

In figure 7 the Product Handling package has been decomposed into smaller components. The “Basket”, “Overview” and “Domain Model” from the original Beer Web Store has been generalized so that they do not contain any product specific handling. A “Payment” package has been created that shall support all needed payment methods. When need for a new payment method arises, this must be added to the package. The supported payment methods for a specific web store must be configured, e.g. at compile time.

5.2 Product Lines - Question 2

Discuss the benefits and liabilities of using a product line approach in relation to the case?

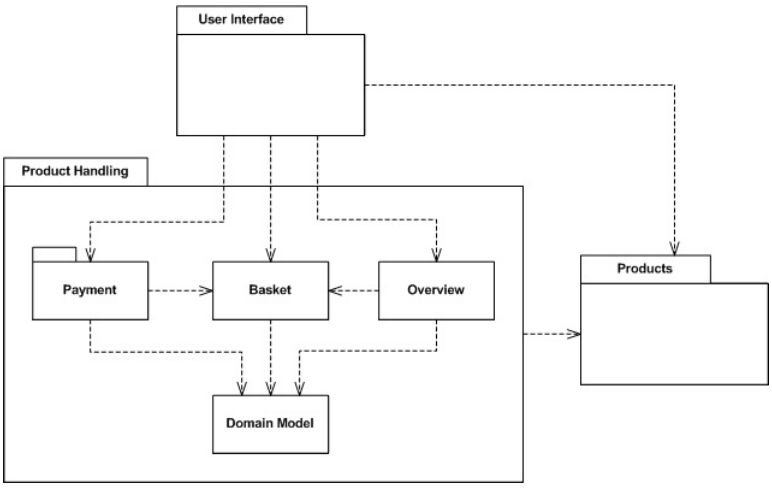


Figure 7: Decomposition of the Product Handling Package for the web store product line

Creating an architecture product line only makes sense if it is believed that it may be used in other products, and the number of products in the pipeline and how well their commonalities may be predicted, helps to determine which approach to use. Naturally the situation is not black and white, as there are many possibilities between the fully evolutionary architecture to the fully initially defined, yet for the purpose of debating advantages and disadvantages the simple black and white is simply easier.

Defining the product line architecture fully up front means a big investment in time and money in order to create the architecture and the first product based on it, so if it is the only product made, then it will go way over budget both in time and money. The idea is that creating the next product will be significantly cheaper, because a very large part of the products functionality is covered by the product line and very little (if any) redesign has to be done. As more and more products are created based on the product line their individual budget savings will help pay for the initial investment, eventually leading to a profit compared to building the products from scratch each time.

Letting the product line evolve means that only the basics are defined up front, only causing a small initial investment, before building the first product. When the second product is to be created more of the product line is defined based on the parts of the first product, which may be used in the second. Naturally the second product is faster to build than the first, due to reuse, yet not as fast as the second product when the entire product line was defined in advance. As more products come along the amount of reuse grow, and would eventually reach the same as the one where everything was defined in advance.

The above two approaches are defined as Heavyweight and Lightweight, respectively. A statistical analysis of these two approaches, compared to simply building the products from scratch, is shown in Figure 8.

Deciding which approach to use, is more than just a matter of choosing between Heavyweight and Lightweight (or fully defined or evolutionary) based on the number of expected products, it is also a matter of how well the common architecture of the future products can be predicted, for if the architecture needs to be changed from product to product then all the advantage is lost.

Based on this the Beer Web Store architect must have decided that the products in the pipe-line and the Beer Web Store share a common architecture and that the future products are so well known that the architecture can be fully specified in advance.

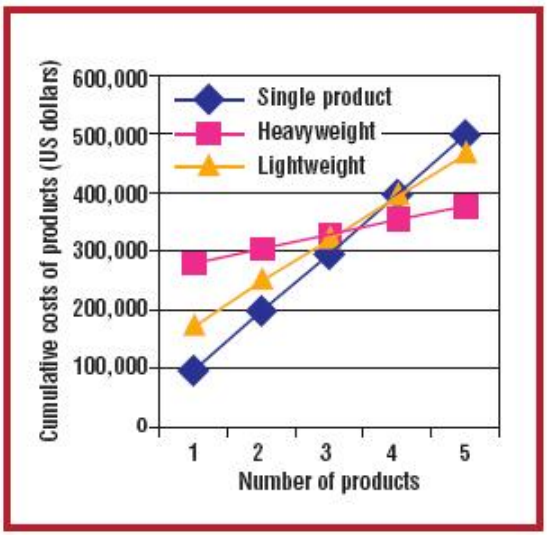


Figure 8: Heavy-weight vs. Light-weight [McGregor et al., 2002]

5.3 Product Lines - Question 3

Relate Service-Oriented Architecture and Product Lines?

As mentioned in Section 3.4 part of SOA is about generalization, just like product lines are. Taking the example given above, having the architecture support mobile phones (i.e. WAB) means that the server must be able to supply a scaled down result of a search as well as the fully blown. The server must thereby handle different types of clients.

The requirement that other product's such as other stamps or soda-pop should be supported, means that the server's domain model and database must support general elements as opposed to just beer elements.

This generalization fits well with SOA, as it is not that clever to make a "Beer purchasing Service", but far better to create a "Purchasing service", which operate on elements. This will allow any company, who has products they wish to sell, to use this service. This is the same generalization requirement that apply to the Product line architecture, where the architecture must handle the purchase of an arbitrary product, be it stamps, soda-pop or beer. It is possible that the architecture requires that it is a quantifiable product, yet this is not an unlikely requirement for a service either.

It may now seem like SOA is the answer to creating a product line, yet this is not the case. It is very possible to create very specific services, like the "Beer look-up service", and this may be just fine for a Beer Web Store based on SOA, yet this is not a product line. The same architectural consideration must be done when creating the elements of a product line and define their interrelations, as when defining the services and their interrelations. The only "structure" that SOA imposes by definition is the loose coupling between the user and the service.

References

[Bass et al., 2003] Bass, L., Clements, P., and Kazman, R. (2003). *Software Architecture in Practice*. Addison-Wesley, 2 edition.

[Christensen et al., 2004] Christensen, H., Corry, A., and Hansen, K. (2004). An approach to software architecture description using UML. Technical report, Computer Science Department, University of Aarhus.

[Kruchten, 1995] Kruchten, P. (1995). The "4+1" view model of software architecture. *IEEE Software*, 12(6):42–50.

[McGregor et al., 2002] McGregor, J., Northrop, L., Jarrad, S., and Pohl, K. (2002). Initiating software product lines. *IEEE Software*, 19(4):24–27.

[Perry and Wolf, 1992] Perry, D. E. and Wolf, A. L. (1992). Foundations for the study of software architecture. Technical report, SIGSOFT Software.

[Sarjoughian, 2002] Sarjoughian, H. (2002). On the role of quality attributes in specifying software/system architecture for intelligent systems. *Measuring the Performance and Intelligence of Systems: Proceedings of the 2002 PerMIS Workshop*.