Global productivity through low coupling and high flexibility

Building a movie rental system in a global team, while minimizing the need for detailed specifications and agreements through modern development principles

Abstract

This project examines how the use of a low coupling, high-flexibility development approach, with a flexible web-service architecture and RESTful principles, can improve the collaboration between groups in distributed, multi-cultural projects. The research is done through a case study of our own project, and a comparison of the findings to a number of case-studies and relevant theory, as presented in the BNDN course literature. The project suggests that such techniques can definitely help teams gain productivity and develop better products together, since the approach forces them to create standardized and well-documented interfaces. As such, the true benefits of the approach are still to be proven, as the collaboration in this project suffered from a number of other problems, including a very tight deadline, that didn't allow the necessary design, testing and documentation before delivering the final web service.

IT-University of Copenhagen, May 2013
Alexander Kirk, Bergar Simonsen, Christian Kuhre,
Morten Hvass & Michael Valentin

Contents

1: Introduction

This report is a part of the course BNDN-2013, at the It-University of Copenhagen (ITU). The project is a joint effort between 5 students at the ITU and 4 students in a related group at Singapore Management University (SMU).

The aim of the project for the ITU group is to develop and deploy a movie rental web service, that the partner team at SMU can client. The SMU team will client the web service, and will develop a graphical user interface (GUI), so that the web service and the GUI together form a functional movie rental platform. The ITU group should also develop their own client, in order to demonstrate the web service produced.

An important aspect of the project is the collaboration between the two groups that are not only culturally different, but also split by 7 time zones. Our project focuses on how we can use standardization and technology to minimize the collaborative problems, and ensure a smooth execution of the joint project.

1.1: Problem

We often see how joint development projects with global collaboration end up as projects that uses more time on working out specifications and agreeing on very detailed design issues when these matters are not actually important from an overall perspective. The endless discussions are often not about improving the user experience or the features of the software, but rather an effort to try and make the rugged endpoints of tailored code come together and have the exact fit that is needed for the various parts of the project to be regarded as successful.

Our project will focus on developing a generic web service that leaves a lot of choices open to the client developers. This approach should minimize the need for very detailed discussions about feature specifications, which are probably best handled in the client developer team. To further improve collaboration, we will use standardized, intuitive, and well documented REST web-services. We believe these to be more flexible and client-developer-friendly than the more tightly coupled SOAP approach.

By combining a generic web service and a standardized interface, we hope to minimize the need for detailed feature specification and instead focus on establishing common principles. Put short, we want to invest our time in establishing a common framework of practices that allow for a high degree of autonomy in the groups, as well as a more effective development process. The key principles, that we think will support our ambition, are low coupling between the server and the client, and a high degree of flexibility in the web service produced.

Besides explaining the considerations and decisions in designing a movie rental web service, our project will examine how the use of these principles for low coupling and high flexibility affects the collaboration and end product.

1.2: Problem statement

"Can establishing a common practice of using low coupling, high flexibility development principles improve collaboration efficiency in a distributed project?"

- Implement a movie rental web service and two clients. One client in a joint effort with SMU and one on your own.
- Describe the implementation and collaboration process, and describe the considerations and decisions with regards to the technical implementation of web service and client.
- Analyze how the use of a generic web service and a REST interface with standardized and well documented methods has affected the group collaboration and the final products of the BNDN project.
- Discuss which advantages and disadvantages the low coupling, high flexibility collaboration strategy implies, and how this could be used to improve collaboration in future projects.

1.3: Scientific method

With regards to method, the project primarily tries to test our hypothesis, about being able to achieve a better product, through introducing a standardized web service and flexible development paradigms. This is done through the project as an experiment, with a qualitative analysis of both the process and the final product, in relation to external sources of theory and experiences on distributed and multi-cultural software development in a more broad perspective.

The scope and situation of the project, along with the fact that we only analyze a single project, makes it harder for us to generalize and make an overall conclusion, but our analysis should help us better understand the dynamics of international collaboration, and the impacts our strategy will have on these. Hence, the main goal of our project is to refine the understanding, and especially our own understanding, of existing theory in the specific area of research.

1.4: Source code

Source code is included on the enclosed CD, and can also be found (including revision history) on GitHub at: https://github.com/BergarSimonsen/BNDN-Project

1.5: Responsibilities

It is a formal requirement for this project that we describe individual responsibilities. All group members have contributed to design and planning, and in this section we will explain other, individual responsibilities.

1.5.1: Alexander Kirk Jørgensen

Alexander's primary responsibility has been the implementation of security, encryption and validation code. This means that he has done our request validation and authentication, including tests of these structures. Alexander has also written some of the entity stubs.

1.5.2: Morten Holm Hvass

Morten's primary responsibility has been the database. He has setup the database and kept the structures up-to-date throughout the project. He has also been implementing the DatabaseConnector and most of the handlers, including a lot of database queries. Morten also wrote a good portion of the unit-tests for the system.

1.5.3: Christian Kuhre

Christian's primary responsibility has been the web service implementation with focus on the controllers and some of the handlers, along with tests for the web service functionality. He was also the one to develop our example client and has been working on a token handler system to make this work properly.

1.5.4: Bergar Simonsen

Bergar's primary responsibility has been the specifications, including use cases and our API specifications. He has also been working on developing some of the controllers and creating web service method signatures. He has also been writing some sections of the report, making most diagrams and illustrations, and did tests on the web service in relation to the SMU deadline.

1.5.5: Michael Valentin

Michael's primary responsibilities have been high-level design, researching literature and writing the report. This means that he has been doing the ER-models, the UML diagrams and some of the high-level system descriptions. He has been writing and rewriting most of the report to ensure consistency.

2: Design and development

This section describes the design and development decisions in our project, with a particular focus on the modeling and architectural decisions that made it possible to create a flexible web service, which allows the client to shape the end product in many different ways.

2.1: Background and theory for this section

This section mainly draws on theory from other courses at the ITU, including Lauesen's course and book on user interface design and specification[i], database theory from BIDD course, and object oriented (OO) design and development as presented in BOSK and BADS courses, with a special focus on the OO-practices described by Bertrand Meyer[ii]. The theory will not be discussed in greater detail, but rather used for references throughout the chapter.

2.2: Domain description[iii]

Developing a custom movie-rental-system can be a challenge. Developers struggle to get the data-model right and the handling of video files, payments, and access controls can be a great challenge. The developers however want to be able to develop a truly custom app, where they, themselves, can control the way users interact and customize all sorts of features to fit their own great ideas.

This yields for a generic and customizable web service, yet we can still identify key concepts that the different services we can think of will have in common:

The client developer wants to be able to upload movies of different formats and to tag these files with all sorts of data and metadata. It is also important that the system keeps track of who uploaded the file and when. It would also be of great use if the system could automatically register statistic data on movie views, etc.

Another important aspect of the movie rental service is the rating and commenting system. End users can help each other get a better experience with the application

by letting each other know which movies are worth watching by giving ratings and reviews to movies watched.

The client developer also needs a way of collecting payments. Payments can be made either on individual movie basis or with a subscription. In relation to the payments, the ability to control which movies can be watched by which users at a given point in time is important.

2.3: Web service requirements and analysis

The project came with a set of mandatory requirements for the web service, as stated in the project description. In order to make our web service as flexible as we wanted it to be, we decided on creating a web service that could support many different movie rental systems, each more specific and customized. This meant building the service as generic as possible, leaving the more specific requirements to the client.

Since the mandatory requirements state that the system must be a rental system, some sort of payment had to be included. We decided to deal with both pay-per-view and subscriptions in a way that allowed our client-applications to have as much flexibility in creating different product offerings as possible. Building our web service in a generic manner also implied creating a more generic tagging system, which allows for some of the mandatory features to be implemented, but also allows for a lot of uses that are not part of the mandatory project requirements.

We also decided on doing a flexible system for handling permissions. The system should handle permissions on both user and group level and leave it relatively simple for the developers to utilize the permissions system throughout the code.

An important part of the project was the negotiation of requirements with our corresponding Singapore Group. The Singapore Group however, was not so ambitious and did not have a lot of demands. This meant that our initial suggestion for web service features covered all their needs, rendering the negotiation of requirements rather trivial.

2.3.1: Limitations and scope

Our project is limited to creating a flexible, functional, and generic web service. Therefore we do not focus as much on the business part, although there is a subscription/pay-per-view element implemented in the system. We give the client developers the ability to handle orders in the system, data-wise, but we do not support any kind of real payment system or handling of real payment transactions.

Neither do we consider any kind of media rendering or conversion. The media is served as it was uploaded to the system, with no consideration of which media types might be supported in different contexts. The project does not deal much with statistics either. Some details can be collected, but it's not the focus of the project.

The common attribute of all the limitations for the project, is that they are not very important for developing a functional system on top of the web service and that they could relatively easily be added to the system later on. Some aspects, such as real payments, conversion, etc. would be extremely important for using our web service in a real life context. For our project though, the abstractions and limitations on these aspects are completely fine; we can still test our hypothesis, as the limitations mentioned do not really affect the way we collaborate.

2.3.2: Final System requirements

Based on the mandatory requirements, the extra features introduced for this project, the limitations and scope for this project, and the use case analysis[1] this is our final system requirements for the movie rental web service:

Non-functional requirements

- The system must consist of a web service and a client
- The system must support at least three tiers of users:
    - Regular users
    - Distributors (access to upload movies to the system)
    - System admin (super user)
- All single entity calls must return a response in less than a second (not taking external factors into account, such as bad/slow internet connection etc.)
- Persistence. The system must still have data available (user data, media data etc) in case the client and/or server breaks down

[1] The use-case analysis can be found in appendix 2 and is inspired by the techniques taught in BDSA and BSUP courses at ITU.

Functional requirements

- Users can sign up and create a personal account on the system
- Registered users must be able to log into the system
- Users must be able to rent and watch media (download or stream)
- Users can pay to watch a single movie
- Users can pay for a subscription, allowing the user to watch all movies for a limited time
- User must have the ability to search for movies based on tags
- Users must be able to post/edit/delete comments and ratings on movies
- Users can read comments/ratings from other users
- Users can have user defined list (such as favorites and playlists)
- Distributors can upload/edit/delete movies. This includes media files as well as meta data (information) about the movies
- Distributors can assign tags to their movies, making them easier to search for
- System admin can send private messages to users
- System admin can give/take away functionality from users
- System admin can block users
- System admin can remove comments/ratings that users have posted
- System admin can remove media

2.4: Web service design

This section describes the system design and architecture in our web service. The section will describe our considerations and explain how our system design and architecture makes the system support our ambitions about functionality and flexibility.

2.4.1: Designing a generic web service

A key part of our project is to make the web service generic and to make it accommodate for as many client app scenarios and features as possible. That is why we have put great effort into designing the data model and web service in a flexible way, which allows the client developer to make most of the decisions with regards to what data should be available and how the system should be used in general.

While the flexibility is very useful from a technical point of view, the approach also gives the client developers more responsibility and requires them to put greater effort into configuring the system to fit their exact needs. Though we tried to make

these configurations simple and intuitive, it is clear that our approach leaves a lot more concepts for the client developers to grasp and a lot more decisions for them to consider.

We did however decide on doing a truly generic and flexible web service. This was both to test our own ability to build such a system, but also to see how well the SMU group of client developers would deal with these concepts.

2.4.2: Using REST as the web service interface

REST is both a set of principles and a protocol that together forms a modern standard for web service interfaces. A REST web service uses standard HTTP-requests, such as GET and POST, and gives readable, minimal responses, typically in JSON or XML. Responses are given in plain text without any mandatory overhead for types or similar. This part of the REST suits our project well, since it is a standardized, accessible and lightweight protocol, which can be accessed from almost all other programming environments and languages.

But REST is more than just a protocol; it also comes with a lot of best practices for what features the web service should have and how it should make them available. A web service is said to be RESTful when it incorporates these principles. One of the principles is that the server should be stateless, and all state information encoded in the requests[2]. This leaves the server without the need to keep track of session states and allows it instead to focus solely on data manipulation and representation. The stateless servers also allows for easier distribution of computing efforts and therefore scalability - for example in a cloud-style server environment. Other principles include a standardized access to entities and loose coupling. This project shall not discuss the deeper aspects of the RESTful principles, but instead refer to Roy Fiedling's doctoral dissertation[iv] for some interesting thoughts on the concepts.

The obvious alternative to using REST would be the more tightly coupled and strongly typed SOAP protocol. This protocol has the advantages of a stronger coupling, more automated type checking, and similar features that can be helpful to the client developer, but requires a heavier and more complex protocol. For our

[2] REST is an acronym for representational state transfer

project and our ambition to develop a flexible and loosely coupled web service, the choice of REST as our protocol and the incorporation of RESTful principles was the obvious choice.

2.4.3: Access Control Lists

An important part of our web service is the Access Control List (ACL) system, which allows the client developers to fine-tune user access both on individual and group basis.

The system consists of a number of actions, defined by the system, that controls if the user can do a certain operation or not. If the action is assigned to the user, the user has permission to do the given action. Permission is granted either for any instance (marked by *) or for a specific instance (indicated by the id of the given instance). For example, if a call is made to delete a comment with id 42, the pseudo communication in the web service logic might look like this:

Model-logic	**ACL-response**
Does the user have the action "DELETE_COMMENT (*)"?	NO
Does the user have the action "DELETE_COMMENT (42)"?	NO
Checks if the author of the comment is the user asking for deletion. It turns out to be so	-
Does the user have the action "DELETE_OWN_COMMENT (*)"?	YES
Goes ahead and deletes the comment	-

If the model logic gets the necessary positive responses from the ACL system, the request is processed without any further actions. If, however, a necessary response is negative, a special exception is thrown to be caught and converted into an error-response with a status indicating that the user does not have the necessary permissions to perform the requested action.

The ACL system is a very flexible and customizable way of controlling access to operations and resources. A simpler system, discussed in the design process for our web service, could be to give every user an access level and for each operation check if the user has high enough an access level to do the specific operation. This approach however, would be hard to maintain when adding new functionality, where one might need to introduce intermediate levels and, because of that, rewrite all higher or lower levels to make room for the new level. This would also mean touching all operations that checks for the given levels, which would be a real disaster. Moreover, there is no easy way to know what access level gives you access to which operations, where the named actions from our ACL are a lot clearer as to what they give access to. Furthermore, the access level approach is not very well suited for giving different users different sets of permissions, which is a very big drawback.

Not only is the ACL system flexible and customizable, but it also allows us to use the same system for controlling users access to content. In our particular system, by adding an "expiration date" to the permissions given, we can use an instance specific permission with an expiration to model the rental of a movie, or make the permission without expiration to model buying a movie. We might also introduce a "WATCH_ALL_MOVIES_WITH_TAG" action that would allow the client developer to provide a subscription where the end-user can watch a specific collection of movies after paying for a subscription.

The fact that we can model user purchases as permissions eliminates the need for our application logic to look into what a specific user has paid for, and instead focus solely on evaluating the user's permissions in relation to the requested action, which is after all a lot simpler. When dealing with purchases and access to restricted operations and content, simple is definitely better, and the separation of concerns that our ACL-system allows is in our opinion a major advantage of our web service design.

2.4.4: Data model for the movie rental web service

In our web service, we want to be able to service a number of different client applications. We considered doing a more advanced data model for collecting all the information from the different applications in one database, with the implied ability to make queries across applications if necessary. The complexity however became over the top, and as effective cross-application queries was not really a necessary feature, we decided to have one database for each application, that could possibly be accessed by multiple clients for that application. That means that our data model is meant for one application, so that the SMU and ITU group could actually have different databases, but with the same structure. The database to be used is determined at call-time, based on the API-key supplied by the client.

The Entity-Relation (ER) diagram below describes the structure of our database. This section will describes our considerations for modeling the database this way, focusing on the more interesting aspects, and leaving out more obvious decisions and structures.

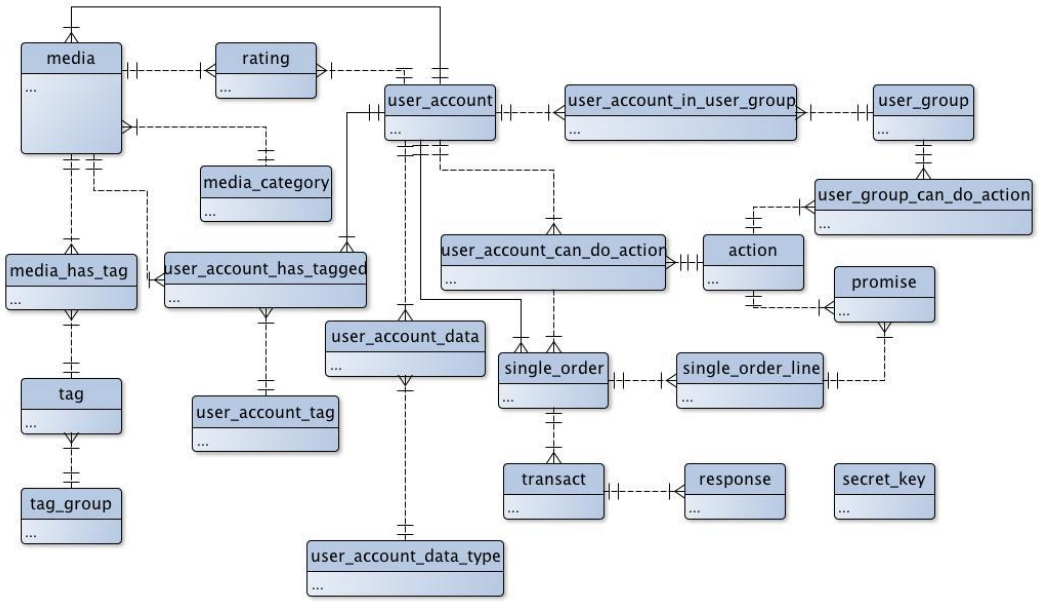

Figure 1: Simplified version of the data model. All fields are omitted for clarity. See appendix 5 for the full data model.

Media is a core entity, representing movies, but potentially also other types of media. Media can be described with tags which are arranged in tag groups. This allows for the creation of a tag group such as “actor”, where “Leonardo di Caprio”

and “Julia Roberts” could be tags that are each related to multiple movies, encoding their appearance in the given movie in a generic and flexible manner. This allow us to query for all movies that have a particular actor, or in more general terms “share certain attributes” in terms of having or not having particular tags. Tags can also be used for “editors pick”, subscription categories, or whatever the client developer demands. This is a very flexible solution for adding and comparing metadata, and is well aligned with our ambition to make the web service as flexible for client developers as possible.

Media can also be connected to a user-specific set of tags (user_account_tag). This means that the client developer can introduce attributions for media on a per-user-basis for storing things like favorites, watch list, etc. This adds further flexibility to the system and opens more possibilities to the client developers.

The user_account is flexible data-wise, as it only has the absolutely necessary data stored. The rest of the information about a given user is stored in a key-value structure where the client developer can define the specific attributes needed in the given client application and assign values for each user. This approach renders listings of users and their data less effective, especially when listing many users[3], but since we are not creating a dating site, or any other system where the listing and search among users and their attributes is crucial, we should be all right.

The user account is connected to the ACL-part of the data model. The ACL-part has a table of actions which can be assigned directly to users, or through the groups that a user is a member of. As described earlier, this structure allows the client developer to adjust permissions in a flexible manner with the possibility to fine-tune permissions both on a user and group level.

The ACL-part is strongly tied to the order-part of the data model, because specific products are modeled as permissions with or without expiration (as described earlier). The promise table encodes a permission that should be granted when the order has been successfully processed. These promises are stored separately pr. order line, to make sure that the purchase history remains intact, even though the

[3] Joining tables is required in queries to collect the user data, making the listings more resource demanding.

client developers decide to change their product offerings. This however eliminates the possibility to easily modify terms (in the form of permissions) for a given subscription or purchase, but we think that a situation where you would want to change the terms for products that are already ordered and paid is quite unlikely.

2.4.5: System architecture

Our system implements a layered architecture, as shown in Figure 2, where a web service class[4] describes the relevant web services, receives the requests and route them on into the relevant controller. The controller will then pass them on to one or more handlers to fulfill the requests. These again use the Database Connector, which is on the bottom of our layered architecture.

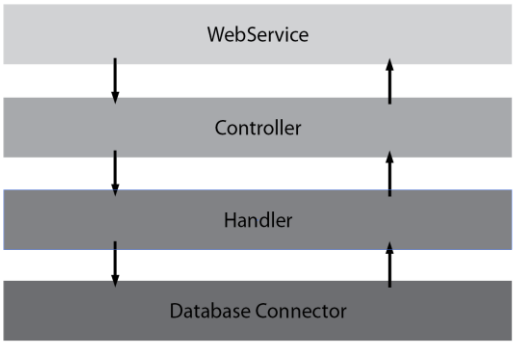

Figure 2: Layered system structure

The separation of concerns between the different layers keeps the coupling to a minimum and makes it easier for developers to edit functionality, as long as it adheres to the original interfaces. The layered architecture also makes it possible to distribute the system over multiple servers. Since each layer is more or less independent of each other, it is possible for each layer to reside on its own server. E.g. the database part could reside on a server with a large amount of RAM for caching results and keeping important data in memory for faster queries.

[4] This is a WCF thing, but it still suits our layered architecture and separation of concerns well

The class hierarchy is illustrated below with an UML[5] class-diagram[6].

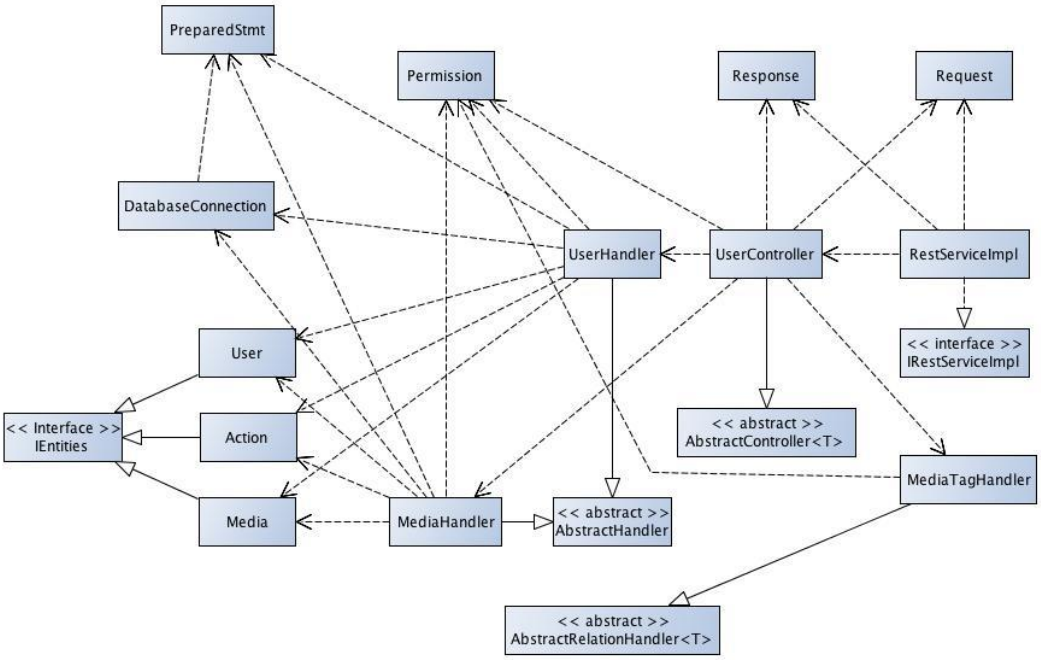

Figure 3: Simplified class diagram. A rich diagram can be found in appendix 3

All calls to the web service are routed through the RestServiceImpl class, which decides which controller to call. The controller interprets the call and populates the request object with the necessary data. It also acquires a permission object based on the user making the call, which it passes on to the relevant handler. The handler now has all the necessary data, interpreted and stored in convenient class-structures. The handler makes the necessary calls to the database and returns its response in form of a response object. The response object contains zero or more entity objects and a call status that the controller can again pass back to the RestServiceImpl.

2.4.6: Prepared statements

As we have not been very consistent in sanitizing input data, we thought it was important to enforce the use of prepared statements to avoid SQL-injections. We

[5] Only one out of many controllers, two out of many handlers and 3 out of many entity classes are shown, in order to make the diagram simpler and more readable.
[6] All methods and fields are omitted for clarity. For a more detailed class diagram, including descriptions of responsibilities for individual classes, see appendix 3.

thought it could be interesting to structure the database interface so that it would be hard, or at least cumbersome, for a model logic developer to query the database without using prepared statements.

Our solution was to make our query and command methods accept only our own custom PreparedStmt object, which is immutable and has a secret signature that only the creator knows of. That way, the DatabaseConnection class can be sure that it has itself prepared the statement (which is therefore safe) if the secret signature is correct.

2.5: Web service implementation

The system was implemented in two stages, one before the SMU deadline and one for the ITU deadline. Unfortunately, the hard time constraint on the first stage meant we had to do a simpler design, very different from the one described in this report. The whole system was basically implemented as only a few classes, and though the system would propose the same interfaces and pass our unit testing, the code behind it was bulky and hard to maintain. The second stage however, left us the chance to implement our design decisions and make the overall system more sound, but also meant that we had to rewrite almost our entire codebase.

2.5.1: Design and implementation

During the implementation we have been using ER and UML diagrams for sketching out and discussing the overall design, and we have updated these diagrams to reflect our current design and act as blueprints for the developers. We have tried to develop together often and practiced pair programming for especially the more complex and important parts of the system. This way of developing worked well for us, and ensured that the entire team headed in the same direction with the project.

2.5.2: Tools and collaboration

We used GitHub as version control and quickly established a Facebook group as an asynchronous communication platform. Both have worked very well, although a few team members who were inexperienced in using Git had some overhead in figuring out how to merge files. For collaborative writing and sharing of

information we used Google Drive, which has worked very well for us. We have tried to keep the number of tools at a minimum to keep things simple and we feel like this approach has really paid off, as the communication and collaboration in the group has been without serious problems. The consistent, weekly meetings[7] helped us take up relevant issues face-to-face, draw stuff to each other on the whiteboard, and grasp the technical concepts together, which was also a great help.

2.5.3: Design changes

We described that we would create a dynamic web service to allow for our clients to build their own custom media rental service and we have stuck with most of our design choices with respect to that. The only thing we have changed is how we handle data on the server and this does not influence the overall flexibility.

Even though we did not change the design along the way, it was not a complete success due to almost starting over with the code after the SMU deadline. This means that the system is not yet fully implemented, and we do not get the full benefits from our design approach. Since we did not have the time to develop the whole system we focused on first meeting the requirements.

2.5.4: Test strategy and tests

To test our software we have decided to combine unit testing and manual testing. Unit testing is used for most public methods, for example the security aspects of the service, while we test the media and database interaction manually using Fiddler[8]. The client testing involves making sure that the client can communicate correctly with the server using the appropriate RESTful methods. This includes testing the login, along with uploading and viewing media on the server.

The service testing is focused on making sure that the server is capable of receiving media and finding said media for future usage. The service should also be able to authenticate clients, authenticate users of the clients, and properly encrypt passwords and login-tokens for future usage. We have unit-tests for the Authenticator, Encrypter, Permissions and TokenHandler.

---

[7] Summaries from group meetings can be found in Appendix 6
[8] Fiddler is an application used for web debugging, and can be found here: http://fiddler2.com/

In general our tests are not very well planned and it is hard to assess how well our tests cover different functionality and scenarios. Besides that, we are pretty certain that there are still more areas of the system to be tested, and if we had more time, we would have made a greater effort to introduce a more structured set of tests.

2.6: Developing an example client

Having focused a lot on our web service, we ended up very short on time for doing the example client. We did however manage to put together an asp.net-based client, which showcases a variety of our web service features. We must admit that the client application is not very well designed, neither engineered, so we will keep this section to a minimum and focus on other parts of the project instead.

We decided to focus mainly on developing the user and media part of the client. This approach was taken because we wanted the client to meet as many of the overall requirements as possible with the time we had. We did not put any focus into making an appealing frontend design, as time did not permit this, and we found other aspects more important. This does not mean that our client is impossible to navigate, but rather that it may not very pretty.

We created a user manual which describes the functionality in the client. The user manual can be found in appendix 7.

3: Process and collaboration

This section focuses on the process and collaboration, in particular how our development efforts have affected the collaboration, and how these observations adhere to our hypothesis. We will only consider the first part of the project where we collaborated with the SMU group.

3.1: Background and theory for this section

This section draws on the BNDN course literature for theoretical concepts used in the analysis and project experiences for comparison. It also draws on a number of other articles regarding more general concepts about distributed development, that are relevant in order to discuss our observations in greater detail, and better relate them to our hypothesis.

3.2: Process

The project description and conditions gave a small number of mandatory and suggested deadlines, which acted well as synchronized milestones[v] among the teams. This allowed us to keep the individual development processes detached, minimizing the need for process compatibility[vi], while still keeping the overall project in sync[vii]. This also played a part in alleviating most of the problems imposed by the temporal distance[viii] between the groups.

For our part the process did not feel very well structured, and even though we did make a rather specific project plan, we did not meet a lot of the deadlines due to uneven workloads in other courses and other external factors. The SMU group seemed to have similar problems, but besides frustration on the individual team, the local problems did not particularly effect collaboration a lot.

The primary problem in the process was that the SMU group had a great deal of trouble using the web service, mainly because they did not have any previous experience from working with REST web services. This meant that we spent a lot of our time trying to teach the SMU group how to utilize the web service in order to

build their client. This in turn made it harder for us to keep up with our own deadlines.

3.3: Communication practices

The first contact with the SMU group was the video conference arranged as a part of the course. It was positive to get a visual on the other group members, but poor sound quality made it very hard to communicate or establish a personal relation to gain trust[ix]. The next couple of meetings were held on Skype, but after having agreed on the overall aspects, communication changed from planned meetings to less formal communication on the Facebook group and via peer-to-peer conversations[x] on Skype.

3.3.1: Facebook group and Skype

All group members on both teams were familiar with Facebook and used the service daily, so Facebook turned out to be an excellent platform for informal and asynchronous communication in a known environment. The group-facility allowed members to ask questions for everyone, eliminating the problem of not knowing who to contact[xi], and the known and widely used platform made the cost of initiating contact reasonably low. The asynchronous aspect of the communication accounted for the temporal distance, but also gave the communication a slower pace, which carried the consequence that not every question would necessarily be answered right away.

That is also why Skype acted as an excellent counterpart. While it required considerably more effort to setup a meeting and open a program that not everyone used on a daily basis, Skype in turn gave a way faster pace in the communication. This meant that more complex questions were sorted out quicker, by letting the groups be able to communicate in a more conversation-like way. Together the two systems formed an excellent communication platform, where most general questions could be asked and answers shared in the group on Facebook, whilst the need for conversations and discussions among relevant team members could be carried out one Skype.

3.3.2: Web service API specifications

In order to make it as easy as possible for the SMU group to understand the web service, and write a good client, we created a complete API specification. Some examples from our specification can be seen here:

Get media with specific id

Url structure:	/media/{ID}
Description:	Get a specific media, based on it's id
Method:	GET
Parameters:	**id** Id of the media to get.
Response:	Response object containing the media object.

Figure 4: Example API spec for the GET media method

Create a new media

Url structure:	/media
Description:	Create a new media and get a path for your upload. This will only create an entry in the database with the meta data provided.
Method:	POST
Parameters:	None
Content-Type	**application/json**
	Media Media object containing all information.
Response:	Response object containing the newly inserted media object.

Figure 5: Example API spec for the POST media method

As seen in Figure 4 and Figure 5, the API specification carries all the information necessary for a client to use the web service. In addition to the examples in Figure 4 and Figure 5, the API specification has a detailed explanation on how the web service works, in terms of response format, error messages, and other attributes that are common for all methods in the web service. For the full API specifications, please see appendix 1.

3.3.3: Challenges

Though we managed to keep the project in sync, and produce an acceptable result together, collaborating with the SMU group proved to be a challenge in several ways. We found that the SMU group would regard us as responsible for everything technical, and ask us a lot of questions regarding how they should connect to our web service, but also how they should build their client. This was a big frustration to us, as we did not feel it should be our job to answer all of these questions, but on the other hand we felt that we had to help the SMU group to progress, so we could

keep the joint project on track. This frustration was intensified by the tight project schedule for the collaborative part of the project.

In general, the time schedule was a problem. The deadline for the SMU group was placed just after our Easter Break, and because most of our group had been away on vacation, there was an overly intense period just after the Easter Break with a lot of overtime work in order to get the web service ready for the SMU deadline. This issue could have been dealt with through better planning between the groups, but we still think that the time frame for the collaborative part was very tight, with respect to developing a fully-featured, tested, and documented movie-rental web service.

3.4: Reducing intensive collaboration

A lot of sources, based on observations, suggest that it is wise to reduce intensive collaboration, especially in projects with big temporal distance[xii] and differing cultures. Herbsleb suggest that we define as small an interface as possible[xiii], with well understood products for each team to work on more separately. Carmel names the reduction of intensive collaboration as his number one tactic and explains how that if teams do not need to share their work too often, less coordination is needed[xiv].

These views support our hypothesis very well, as they define the benefits of reducing intensive collaboration in a more general way. This means that if we are actually successful in reducing the amount of collaboration by providing clear interfaces and early division of labor, we should also get benefits in terms of a smoother process and a better product in the end.

3.5: Cultural aspects

There were clear cultural differences between the groups, but not only the traditional ethnically related cultural differences. The fact that our educational background and technical skills differed a lot proved to be a problem. The Singaporean team members came from a management university and did not have a lot of technical IT-courses, whilst the Danish team members had a more technical

background, but not the same competences in relation to business and management as the Singaporeans.

The differences in the educational background showed mostly when the two teams dug into more technical stuff, where the Danish group had a couple of incidents where the Singaporean group agreed on some technical details, e.g. regarding the REST protocol and principles that they did not actually understand. This does, of course, also touch upon some of the cultural dimensions[xv], but the educational background definitely played an important role in this matter. We feel that the Danish group could have been a lot more aware that they were talking to a group with a less technical background than they have been used to from their projects at the ITU.

The project also showed some indications of the more traditional cultural aspects, as in the example above, where Hall's 5th cultural dimension[xvi] (Agreement) showed as an obvious problem in the collaboration. We saw more of the classical, cultural differences, but most of them were quite obvious and as such, easier to deal with.

The last dimension however, that we will touch upon in this project, is Hofstede's 1st on hierarchy[xvii]. It quickly became clear that the SMU group regarded ITU group as superior on the technical part, which was the most important part of the collaboration. For the Danish group, it would still be natural to give feedback and inputs, even though we felt somewhat less knowledgeable of the topic at hand, but for the Singaporean group this imposed a different situation. They placed themselves in a less dominant position, and were more likely to ask questions and seek advice than give suggestions or put up demands. This also became a problem around the SMU deadline, as the SMU group did not really make their needs clear early on. This miscommunication is of course also a result of the chaotic situation on both parts, with other course work during the project and busy student schedules, but we believe that the submissive position that they ended up in made things even worse.

3.6: Ethnocentrism and collaborating across cultures

It was a problem in the collaboration that we ended up as "supervisors" for the SMU group, as opposed to two groups collaborating. Cramton's research[xviii] can help us explain what happened, as the situation was a great example of latent faultlines[xix] in the subgroup dynamics. The faultlines regarding the differences in educational background became salient with the initial discussions about architecture, protocols, and data models as the activating event[xx].

We think it is clear that our approach has been too ethnocentric[xxi]. Not in the traditional way; we have done a lot to try and eliminate the cultural barriers, and with decent results, but with regards to the educational background. We were not aware that this was actually an important, latent faultline between the two groups, and that we should have taken greater precautions. A tendency to blame problems on the less technically capable Singaporean group appeared among the Danish group early in the process. The resulting rhetoric and behavior in our collaboration with the Singaporean group have had negative effects on working together in equal collaboration.

Cramton suggests that partly positive results, which are actually well aligned with the intentions for our project, can be generated with an ethnocentric approach. In our case however, the combination of ethnocentrism and hierarchal cultural dimensions of the collaboration turned out to have a negative impact. Should we do a similar project, we would be even more careful to have a more ethnorelativistic[xxii] approach and try to establish a more equal form of collaboration that way.

4: Discussion

It is relatively clear, that while our approach to the development process has produced a decent final result, it has not been a success in terms of collaboration. The SMU group simply did not seem to grasp the concepts we were trying to introduce and we ended up in a role as supervisors rather than co-workers collaborating on the project.

Moreover, the time pressure in connection with the SMU deadline lead to the release of a web service that was not thoroughly tested and not nearly as well documented as it was supposed to be. Even though these circumstances had been different, we are not sure the project would have went better, since the SMU group did not possess the technical background to truly benefit from the structures that we delivered to them. On the other hand, had we been more aware of the latent faultlines at question, we might have been able to overcome more of the problems through a better distribution of responsibilities and a more collaborative style of working among the groups.

Besides the complications with the SMU part of the project, our approach turned out to have other positive impacts on our design and architecture in the system. In particular, we feel that the constraints that the REST methodology implied, was the inspiration for some good design decisions in our project, and a very nice web service to client relationship in our part of the project.

This indicates a partial success for the utilized principles. The success is mainly based on our own subjective opinion that our service came out more well designed than it otherwise would have, but the advantages of this particular style of design is well documented[xxiii] and accepted among many. The success is only partial though, because it seemed to introduce more confusion for SMU group than it helped them deliver a better product. We think this is mainly due to the tight deadline and the SMU groups lacking technical background, but it is definitely also a question of our missing focus on collaborating in a more egorelativistic style, rather than the principles themselves. These circumstances however, render our experience of collaboration from this project rather useless with respect to testing the principles influence on the collaboration.

5: Conclusion

Although it is difficult for us to answer our problem statement, based on this project, we do see some interesting results. It is evident from the literature and our own experience in the project, that the establishment of well-defined interfaces, with as small a point of contact as possible, reduces intensive collaboration and helps work go more smoothly. In addition to that, we have realized that it is not only the technical aspects that can benefit from these characteristics; the collaboration can do the same. We believe that our project could have had better success, had we been more aware of the latent faultlines earlier on and defined some clearer roles and responsibilities among the teams. This synchronization is well aligned with Conway's law[xxiv], and based on the experience from the project, we believe that keeping this in mind can greatly help improve the success in a multicultural distributed project with big temporal distance.

With regards to testing our original hypothesis, it could be very interesting to see how another, more technically skilled group, would work with our web service, which is now more stable, tested, and documented. Another such example would possibly help support our claims better about the advantages of the development style in terms of collaboration and pay some more respect to the relevant concepts as a whole.

6: Bibliography

Carmel, E., & Agarwal, R. (2001). Tactical approaches for Alleviating Distance in Global Software Development. *IEEE Software, 18*(2), 22-29.

Cramton, C. D., & Hinds, P. J. (2005). Subgroup Dynamics in Internationally Distributed Teams: Ethnocentrism or Cross- National Learning? *Research in Organizational Behavior, 26*, 233-265.

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. UNIVERSITY OF CALIFORNIA, IRVINE.

Herbsleb, J. D., & Grinter, R. E. (1999). Splitting the Organization and Integrating the Code: Conway's Law Revisited. *ICSE*, 85-95.

Herbsleb, J. D., Paulish, D. J., & Bass, M. (2005). Global Software Development at Siemens: Experience from Nine Projects. *ICSE*, 524-533.

Holmstrøm, H., Fitzgerald, B., Ågerfalk, P. J., & Conchúir, E. Ó. (2009). Agile Practices Reduce Distance in Global Software. *Information System Management*(23), 7-18.

Lauesen, S. (2005). *User Interface Design - A Software Engineering Perspective* (1st ed.). Addison-Wesley.

Meyer, B. (1997). *Object-Oriented Software Construction* (2nd ed.). Santa Barbara, California: Prentice Hall PTR.

Olson, J. S., & Olson, G. M. (2003-2004). Culture, Surprises in Remote Software Development Teams. *ACM Queue, December/January*, 52-59.

Paasivaara, M., & Lassenius, C. (2003). Collaboration Practices in Global Inter-organizational Software Development Projects. *Software Process Improvement and practice*(8), 183-199.

Sliger, M., & Broderick, S. (2008). How will I work with Other Teams Who Aren't Agile". In *The Software Project Manager's Bridge to Agility* (pp. 233-248). Indiana: Addison-Wesley.

7: Appendices

The following section contains all of our appendices for the project, including diagrams, descriptions, meetings notes and other relevant artifacts from the project

Appendix 1 : Web service API Specifications

This appendix contains our final web service API specifications, in the format which we found to be most useful in communicating the API details to the SMU group.

Appendix 2 : Use cases

This appendix contains our use case analysis, that served as the foundation for making our system specifications

Appendix 3 : Class diagram

This appendix shows our final class diagram in UML format.

Appendix 4 : Project Plan

This appendix shows our overall project plan, which we presented to the Singaporean group, and used for coordinating milestones in the project.

Appendix 5 : Data model

This appendix shows our data model, represented by an ER-diagram

Appendix 6 : Summary of group meetings

This appendix contains summaries from most of our group meetings. The notes have been translated into English to match the language of the report, but are included as they were written during the project, and some parts might be harrd to understand without the relevant context.

Appendix 7 : User manual

This appendix contains the user manual for our example client system.

Appendix 8 : Review of group 5 preliminary report

This appendix contains our feedback on the preliminary project report, which we received from group 5. The written feedback is accompanied by a scan of the report from group 5, with hand-written highlights and comments.

Appendix 9 : Review of our preliminary report

This appendix contains the feedback that we got on our preliminary report from group number 7.

[i] (Lauesen, 2005) with special focus on chapters 4 and 5

[ii] (Meyer, 1997) with special focus on the section B & C

[iii] (Lauesen, 2005) Chapter 5

[iv] (Fielding, 2000) Especially chapters 5 and 6

[v] (Paasivaara & Lassenius, 2003) p.187

[vi] (Herbsleb, Paulish, & Bass, Global Software Development at Siemens: Experience from Nine Projects, 2005) p. 527

[vii] (Sliger & Broderick, 2008) p.240

[viii] (Holmstrøm, Fitzgerald, Ågerfalk, & Conchúir, 2009) p. 11

[ix] (Herbsleb, Paulish, & Bass, Global Software Development at Siemens: Experience from Nine Projects, 2005) p. 531

[x] (Paasivaara & Lassenius, 2003) p. 189

[xi] (Herbsleb & Grinter, Splitting the Organization and Integrating the Code: Conway's Law Revisited, 1999) p. 90

[xii] (Holmstrøm, Fitzgerald, Ågerfalk, & Conchúir, 2009) p.11

[xiii] (Herbsleb & Grinter, Splitting the Organization and Integrating the Code: Conway's Law Revisited, 1999) p. 94

[xiv] (Carmel & Agarwal, 2001) p. 24

[xv] (Olson & Olson, 2003-2004) p. 54

[xvi] (Olson & Olson, 2003-2004) p. 54

[xvii] (Olson & Olson, 2003-2004) p. 54

[xviii] (Cramton & Hinds, 2005)

[xix] (Cramton & Hinds, 2005) p. 245

[xx] (Cramton & Hinds, 2005) p. 246

[xxi] (Cramton & Hinds, 2005) p. 235

[xxii] (Cramton & Hinds, 2005) p. 240

[xxiii] (Fielding, 2000) Especially chapters 5 and 6

[xxiv] (Herbsleb & Grinter, Splitting the Organization and Integrating the Code: Conway's Law Revisited, 1999) p. 85