Hand in H6

Software Architecture in Practice

Book Swap Case

Department of Computer Science, University of Aarhus

Aabogade 34, 8200 Århus N, Denmark

Gruppe: Bravo

20074842, Lars Kringelbach, lars@kringelbach.com

20064684, Marjus Nielsen, Marjus.nielsen@gmail.com

20074877, Morten Herman Langkjær, morten.herman@gmail.com

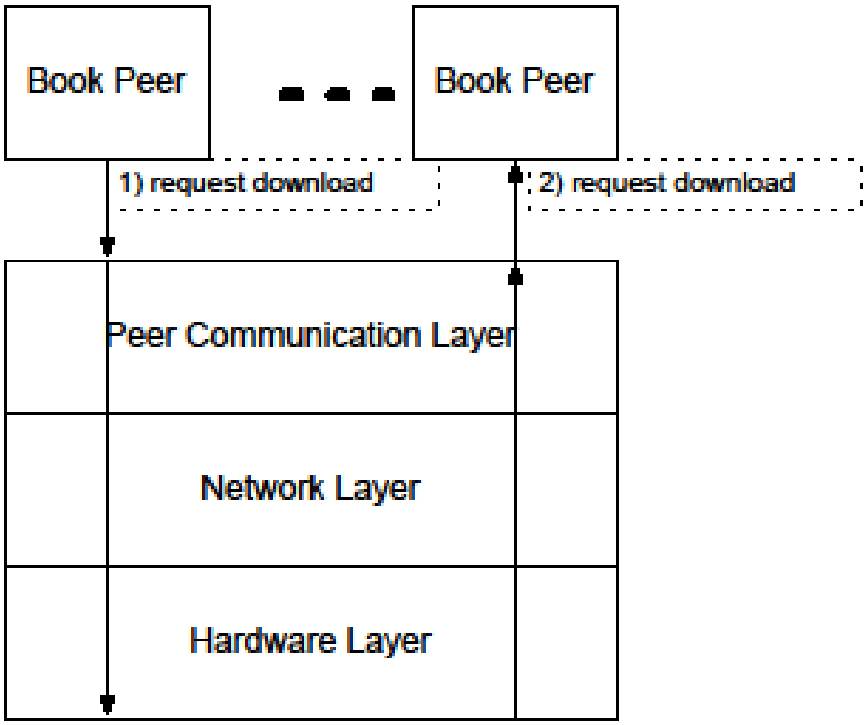
20054680, Peter Madsen, pm@chora.dk

<<Dato: 27. maj 2008>>

Indholdsfortegnelse

1 Introduction

"'Book Swap' is to be a new peer-to-peer (P2P) system for swapping books among "peers". Here peers are distributed nodes that acts as both clients and servers. As a part of a business case for Book Swap, the software architect has outlined the following software architecture":



"The main idea is that a Peer controls a set of books that other peers may download, that peers are equal, and that there is a Peer Communication Layer that abstracts away details about a specific network".

2 Besvarelse

2.1 Architectural Prototyping

"The architect has decided that the system should be implemented in Java and that the following interface should be observed when implementing Book Swap":

```
public interface Peer {

            /**
            * Download a Book based on a keyword search
            */
            public String download ( String keyword );

            /**
            * Upload a book to this Peer
            */
            public void upload ( String book );

}
```

Now, the software architect wants to apply the technique of architectural prototyping.

Please consider the following:

- Outline which type(s) of architectural prototyping that is/are relevant in this setting
- The software architect is concerned about whether to use RMI or raw TCP/IP (socket) communication in the prototype. Describe how architectural prototyping may help the architect in making this decision
- The software architect is concerned about modifiability. Describe how you would create an architectural prototype in this case
- Discuss what benefits and liabilities there is of using ArchJava for architectural prototyping in this case
- The software architect decides to make a Service-Oriented Architecture. Discuss this solution

"Outline which type(s) of architectural prototyping that is/are relevant in this setting":
Der findes tre forskellige typer af arkitektoniske prototyper

Type af prototype	Karakteristika
"Exploratory"	At udvikle arkitektoniske løsningsforslag til et system
"Experimental"	At evaluere arkitektoniske løsningsforslag af en arkitektur
"Evolutionary"	At videreudvikle et system eller en prortype til at vise ændringer af en arkitektur.

Da formålet med prototypen er at vise hvorledes et forslag til en arkitektur vil fungere, vil det være oplagt at lave en exploratory prototype som kunne teste hvor vidt arkitekturen vil virke med en bestemt teknologi, og i et bestemt setup. Formålet med prototypen kunne være flere, f.eks. hvorledes registrering og identifikation af "peer"s i systemet skal udvikles, samt f.eks. performance af de valgte teknologider til at uploade og downloade bøger.

"The software architect is concerned about whether to use RMI or raw TCP/IP (socket) communication in the prototype. Describe how architectural prototyping may help the architect in making this decision":

Ved at opstille nogle målbare kriterier for hvordan systemet skal opføre sig, f.eks. ved hjælp af kvalitets attribut scenarier jvf. vil det være muligt at lave en prototype som benytter RMI som kommunikationsmedie, og en anden prototype som benytter rå TCP/IP Socket kommunikation. På den måde vil det kunne testes både hvor besværligt de enkelte teknologier er at benytte i forhold til udviklingsprocessen, men der vil også kunne måles på det overhead der sandsynligvis vil være i at bruge en objektorienteret netværksteknologi, frem for eksempelvis en propreitær binært serialiseret komprimeret rå datastrøm.

Yderligere kunne prototypen også have den sideeffekt at den kunne vise hvor enkelt det vil være at skifte netværkslaget ud, og dermed sige noget om modifiability.

"The software architect is concerned about modifiability. Describe how you would create an architectural prototype in this case":

I forbindelse med udviklingen af en prototype som omhandler vedligeholdbarhed, og modificerbarhed, vil det være oplagt at forsøge at definere på hvilke niveaer der ønskes denne modificerbarher. Skal der f.eks. kunne udskiftes netværkslag, eller er det systemets brugergrænseflade der skal kunne ændres eller skiftes ud. Igen kan dette gøres ved at fastlægge de kvalitetsattribut scenarier, som er relevante for modificerbahed. Herefter kunne der laves en prototype som belyser disse aspekter, f.eks. ved at lave abstraktioner over netværkslag, eller benytte MVC patterns. Yderligere vil det være oplagt at kigge på dependency injection, hvor der findes et antal frameworks til f.eks. at understøtte at konfigurationen linkes

sammen på runtime, og ikke compile time, således at man ved hjælp af xml kan stykke sin applikation sammen. Eksempler på dette kunne være JAVA Spring, eller Microsoft XAML(Extende Application Markup Language) . Herefter vil der f.eks. kunne laves en evolutionær prototype med nye krav til netværk eller GUI, baseret på den exploratory eller experimental prototype som ville implementere et forslag til hvordan arkitekturen skulle se ud hvis modifiability er vigtigt.

"Discuss what benefits and liabilities there is of using ArchJava for architectural prototyping in this case":

Arch JAVA er en udvidelse til java som via syntaktiske udvidelser understøtter components og connectors. De åbenlyse fordele der er ved at benytte disse udvidelser er at deres syntaks understøtter en umiddelbar mapning fra C&C view til kode, og omvendt. Dermed er det muligt på enkel vis at lave en hurtig konvertering af kode til viewpoint, og dermed kunne benytte sine arkitektoniske prototyper som driver til dokumentationen af arkitekturen.

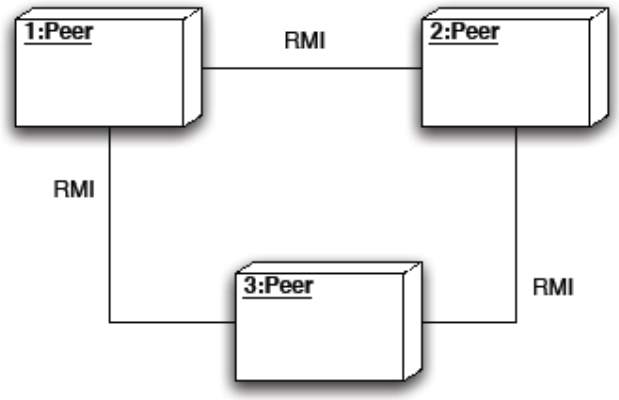
En åbenlys ulempe ved at benytte Arch JAVA, vil være at når der lægges et ekstra lag af abstraktion ind over koden, lægges der samtidig et ekstra lag af kompleksitet på. Der vil være behov for at de personer der skal udvikle i ArchJava, skal trænes i at bruge teknologien, og hvis der ændres på bemandingen af et projekt over systemet livscyklus, vil der være behov for at disse personer skal læres op i endnu en teknologi. Yderligere skal der tænkes langsigtet i forhold til teknologiholdbarhed,, hvor det kan være der bliver behov for at understøtte e nyere version af Java, men hvor der ikke er udviklet en tilsvarende ArchJava til. Det vil så betyde at den allerede producerede kode ville skulle skrive om, eller man skulle blive på den eksisterende Java platform. Dermed vil en prototype skrevet i ArchJava kunne få langsigtede implikationer, hvis ikke det besluttes at bygge den endelige software direkte ovenpå prototypen istedetfor at lave software fra bunden efter evaluering af prototypen.

"The software architect decides to make a Service-Oriented Architecture. Discuss this solution":

Det at lave en service orienteret arkitekturløsninge for systemet lyder umiddelbart som en rimelig beslutning. SOA systemer vil i høj grad udvikles om meget løst koblede systemer, og derfor kunne hver "peer" tilbyde et antal services som andre "peers" kunne benytte sig af. Ved at tage SOA ind i løsningen, kan man så overveje om man også vil gøre løsninge åben for integration fra andre systemer, ved at benytte sig af SOA Reference modellen fra [MacKenzie et al., 2006]. Dermed kræves der også en ekstra indsats for at dokumentere de services der findes i systemet, deres semantik, hvorledes der skal interageres med dem, sikkerhedsprotokoller, indvirken på omverdenen og hvilken kontekst servicen eksekveres i. Valget om at benytte SOA, trækker demed en række andre problematikker med sig, men hvis der er behov for f.eks. det løst koblede system, eller mulighed for at kunne integrere fra tredjepartsprodukter, vil det være et udemærket valg.

2.2 Architectural evaluation

The software architect has designed a deployment structure of the system as shown below:



Here it has been decided to implement the system in Java and to use Java RMI for distributed communication. The main architectural drivers for the system are modifiability and testability and the following is the main use case for the system:

"The user logs on Book Swap as a Peer and offers his books for download. After discovering other Peers, the user downloads books from other Peers and allows these to upload books. The project stakeholders now want to do an evaluation of the system and decide that the ATAM method should be applied."

Please consider the following:

- Describe the steps of the ATAM method and how they would map to the case
- Point to possible examples of sensitivity points, trade-off points, and risks
- Discuss strengths and weaknesses of ATAM in relation to the case
- Discuss in which way DiscoTect could be used to evaluate the architecture of a (prototype) implementation of the system
- Compare and contrast ATAM, architectural prototyping, CBAM, and aSQA as ways of evaluating a software architecture

“Describe the steps of the ATAM method and how they would map to the case”:

I ATAM er der ni skridt der skal gennemføres ad to omgange. Nedenfor er listet vilke skridt det drejer sig om samt hvordan det refererer til dette dette system.

Seance nr:	Processskridt	Book Swap
1	Evalueringsleder præsenter metode for teamet	Ingen bemærkninger.
1	Præsenter business drivers	Fremlæg funktionelle krav omkring udveksling af bøger og peer lookup. Fremlæg relevante kvaliteter og deres samspil, f.eks. performance, modifiability mm. Fremlæg forretningsmål for produktet, hvad er markedet, hvem skal bruge det mm. Præsenter stakteholders, f.eks. managers, kunder, udviklere, boghandlere mm.
1	Præsenter eksisterende arkitektur	En arkitekt, eller anden teknisk kyndig person hvis der ikke findes en arkitekt, fremlægger den eksisterende arkitektur med fokus på view, og hvordan disse imødekommer business drivers.
2	Identificer arkitektoniske tilgangsvinkler	Dokumentér arkitektoniske stilarter, og taktikker,så som SOA, og f.eks. heart beats så registry kan overvåge om en peer forbliver aktiv. Match de valgt taktikker, stilarter osv. med de tidligere præsentere kvaliteter.
2	Generer Quality attribute utitlity tree	For hver af de udvalgt kvaliteter defineres på overordnet niveau de scenarier der skal tages hensyn til i evalueringen, f.eks. performance og tilhørende valg af netværkslag samt modifiability og evnen til at udskifte netværkslaget. Herefter prioriteres de enkelte scenarier efter vigtighed for system og hvor svære de er at opnå.
2	Analyser arkitektoniske tilgangsvinkler	Identificér hvorledes hvert højprioritetsscenarie er implementeret eller imødekommet af arkitekturen, f.eks. performance over netværket ved valg af TCP/IP netværksimplementation ved brug af sockets. Dette gøre bl.a. ved at definere Sensitivity points, tradoff points og risks. Dermed kunne et eksempel være at et sensitivitypoint var valget af TCP/IP som netværkslag. Tradeoff point kunne være at det kunne blive sværere at

		udskifte netværkslaget ved at vælge en et netværkslag på lavt abstraktionsniveau. Risk kunne være at det kunne være uacceptabelt ikke at kunne benytte sprogunderstøttet marshalling og unmarshalling af data på tværs af netværket.
2	Brainstorm og prioriter scenarier	Igen videreudvikles scenarierne, og reprioriteres efter større grupper af interessenter, og dermed forfines og udvides input til systemets kravbillede. Det kunne f.eks. være at det blev besluttet at det skal være muligt at integrere til systemt fra andre programmeringssprog end Java.
2	Analyser arkitektoniske tilgangsvinkler	Igen kikkes der på hvorledes arkitekturen og de beslutninger der er taget tidligere imødekommer de definerede kvalitetsattributscenarier. Dette kunne f.eks. være at kigge på hvordan integration med andre programmeringssprog var understøttet af systemet.
2	Præsenter resultatet af ATAM	Alle de opsamlede informationer præsenteres, både dokumenteret arkitektur, scenarier, utility træer, sensitivity points og tradeoff points mm.

"Point to possible examples of sensitivity points, trade-off points, and risks":

Se eksempler i tabellen ovenfor.

"Discuss strengths and weaknesses of ATAM in relation to the case":

Fordelen ved at bruge ATAM er at den indrager i bedste fald alle interessenter, og er meget grundig. Det betyder at der vil være en stor sandsynlighed for at det faktiske resultat af metoden/processen vil være nok valid information til at arbejde videre med systemet, samt definere hvilke dele af systemet der har størst behov for at blive ændret.

"Discuss in which way DiscoTect could be used to evaluate the architecture of a (prototype) implementation of the system":

DiscoTect er et system til at autogenerere arkitekturbeskrivelser for et kørende system. Det basserer sig på at der er installeret et antal kørende prober I systemet, som aktiveres når dele af systemet eksekveres. Dermed har det mulighed for at give feedback på hvor ofte dele af systemet kaldes, og med hvilken type data. Det vil give mulighed for automatisk at generere dokumentation for, hvor vidt prototypen overholder de arkitektoniske tanker og dokumentation som lå til grundlag for prototypen. Dermed giver det mulighed for at holde den faktiske kørsel af prototypen op imod den udviklede arkitektur for prototypen, og dermed også mulighed for at verificere at prototypen viser det den havde til formål at vise.

“Compare and contrast ATAM, architectural prototyping, CBAM, and aSQA as ways of evaluating a software architecture”:

Hvor ATAM er en relativt grundig og omfattende process, som både er omfangsrig I personer, men også i kalendertid, så er arkitektoniske prototype en metode som skaber hurtig feedback, og et garanteret validt feedback. Ved at lave implementation, kan det nemt identificeres hvor vidt en teknologi, stilart eller taktik har positiv eller negativ indflydelse på systemet. Evalueringstyper der har karakter af spørgsmål, kan blive både tænkte, og risikere at være ufyldestgørende, hvis interessenterne glemmer nogle væsentlige detaljer. Der vil en prototype sørge for at man ”bliver ramt af virkeligheden”, og dermed ikke glemmer detaljer om f.eks. miljøet hvori systemet kører. CBAM og aSQA er begge letvægtsmetoder, hvor CBAM forholder sig til COST/BENEFIT, og dermed hvilke features, eller valg af stilarter/taktikker/mm. Der vil give det bedste afkast basseret på investeringen. Dette kan være en glimrende måde at prioritere bestemte typer opgaver, samt se hvorledes der eventuelt vil være synergi imellem at lave flere opgaver samtidigt. CBAM kan også vise hvor vidt der vil være afhængiheder imellem de forskellige typer opgaver, og dermed også om en implementeret opgave vil påvirke en anden opgave negativt. aSQA fokuserer i langt højere grad på komponenter, og løbendene evaluering af systemets tilstand. Det betyder at den udledte information kan give indblik i hvor der vil være mest værdi i at lægge sine udviklertimer, i forhold til komponenternes tilstand, og deres mål. Alle metoderne belyser systemudvikling, og arkitekturudvikling fra forskellige vinkler, og dermed supplerer de hinanden fint, og kan benyttes i forskellige situationer.

2.3 The Architecture process and Architecture competences

The organization that develops the Book Swap application is a large publishing house that plans to offer its publications for peer-to-peer swapping with each swap incurring a fee for the organization.

The architect is now targeted with presenting a proposed software architecture process for project management.

Please consider the following:

- Outline which architectural design activities are relevant in this case and what they should contain.
- The architect decides to use a backlog in his architecture work. Describe how the backlog may be used.
- Consider which types of duties, skills, and knowledge are most relevant in this case.
- Give examples of how Organizational Coordination and Organizational Learning could ideally take place in this organization.
- Give examples of what would make the organization architecturally competent.
- Give examples of what would make the organization architecturally incompetent.

“Outline which architectural design activities are relevant in this case and what they should contain.”:

For at diskutere relevante designbeslutninger er det nødvendigt at have viden om den organisation hvor i arkitekturprocessen skal passe. Da organisationen er et stort forlag, må man forvente at deres hovedekspertise ikke ligger inden for udvikling af software. Det tænkes derfor at ledelsen ikke er software kyndig og at udviklingen af Book Swap skal forestås af en mindre udviklingsgruppe. Der bør derfor fokuseres på letvægtsmetoder, som er letforståeligt for ”ikke software kyndige”.

Med udgangspunkt i [Hofmeister et al, 2007] laves et skema for en foreslået arkitektur proces med aktiviteter og artefakter for de tre faser, analyse, syntese og evaluering:

	Artefakter	**Aktiviteter**	**Teknikker og værktøjer**	**Beslutningsrationale**
Arkitektonisk analyse	Hoved use cases Hoved kvalitetsatributter	Find hoved use cases ud fra markedsanalyse QAW	-QAS -Use cases	Letvægtsmetoder tilpasset en lille organisation. Fokus på output som er forståeligt for ”ikke software kyndige
Arkitektonisk syntese	Viewpoints Prototype(r).	ADD Arkitekturprototyper Funktionsprototype	UML Skeletapplikation	Letvægtsproces. Arkitekturprototype giver gode muligheder for at undersøge om forskellige kvalitetskrav er overholdt. Prototypen kan desuden tjene som basis for den videre udvikling. Der er desuden tilføjet en funktionsprototype, hvilket ikke direkte er en arkitektonisk aktivitet, men som tjener til at animere hoved use cases over for management, således det så tidligt som muligt kan valideres at er det den ikke software kyndige ledelse forestiller sig.
Arkitektonisk evaluering	Utility grafer Return On Investment System Health System Fokuspunkter Prototype kommentarer	Udfør CBAM Udfør aSQA Evaluer arkitektoniske prototyper	ASQA,CBAM	Letvægtsprocess, lille projektorganistation. Output tænkes let forståeligt for ledelsen, da der er tale om output som minder om det der fås fra fx. økonomiske analyser.
Overordnede proces drivere	Backlog	Opdater backlog	Wiki	Letvægtsproces, kun en arkitekt. Giver en nem mulighed for at dokumentere og have historik på designbeslutninger.

Som det ses af ovenstående skema er der valgt en række letvægtsteknikker, som passer ind i en lille organisation. Desuden er er fokus på at ledelsesrelevant information produceres i et format som er letforståeligt for personer uden erfaring med udvikling af software. Viewpoints og arkitektoniske prototyper er direkte rettet mod udviklingsteamet, og det giver derfor ikke anledning til bekymring at der her er tale om artefakter som ikke umiddelbart kan forventes forstået af ledelsen.

“The architect decides to use a backlog in his architecture work. Describe how the backlog may be used.”:

Det er som skrevet allerede foreslået at der benyttes en backlog. Backlogen benyttes til at beskrive alle arkitektoniske valg (der bør benyttes et veldefineret skema, som vist i H5), udeståender, resultatet af QAW osv. Der tænkes brugt en Wiki, således tiden bliver brugt på indhold og ikke så meget på form.

Backloggen kommer hermed til at indeholde historikken for den arkitektoniske udvikling af projektet, og lige så vigtigt den historik der ligger bag forskellige beslutninger. Desuden vil den tjene som en elektronisk notesblok hvor i arkitekten kan se og notere ”open issues”, således disse ikke går tabt.

Ud over den før omtalte sporbarhed, giver en elektronisk backlog også mulighed for at mere end en person kan deltage i det arkitektoniske arbede.

“Consider which types of duties, skills, and knowledge are most relevant in this case.”:

Følgende er de typiske pligter for en arkitekt (bearbejdet fra [Bass et al., 2008b]):

Generelt område	Specifikt område
Arkitekturarbejde	Lave en arkitektur Arkitekturevaluering og analyse Dokumentation Transformere eksisterende system Andre arkitektoniske opgaver
Ikke arkitekturspecifikke opgaver	Kravspecificering Kodning Test Fremtidige teknologier Valg af værktøjer og teknologier
Interaktion med stakeholders	Generel interaktion Kunder Udviklere
Ledelse	Projektledelse Personaleledelse Supportere øvrig ledelse
Relateret til organisation og forretning	Organisation Forretning
Lederskab og teambuilding	Teknisk lederskab Team Building

Arkitekturarbejdet er naturligt en af arkitektens vigtigste pligter, da der kun er vedkommende til at skabe arkitekturen (transformation af et eksisterende system er naturligvis ikke vigtigt, da et sådant ikke findes).

Da der ikke findes nogen udviklingsorganisation i forvejen, er der i dette tilfælde andre meget vigtige opgaver for arkitekten at udføre, herunder valg af værktøjer og teknologier, interaktionen med stakeholders og især teknisk lederskab og team building, som er nødvendigt for at få den nye afdeling og den nye arkitekturmodel op og stå.

Der må forventes at være en decideret projektleder tilknyttet projektet, hvorfor dette ikke tænkes at være en af arkitektens pligter.

Følgende er en liste af de egenskaber en arkitekt bør være i besiddelse af:

Generelt område	Specifikt område
Kommunikationsevner	Eksterne kommunikationsevner Generelle kommunikationsevner Interne kommunikationsevner
Interpersonelle evner	I teamet Uden for teamet
Arbejdsevner	Lederskab Effektivt håndtere arbejdsbelastning Håndtere information
Personlige evner	Personlige kvaliteter Håndtere ukendte faktorer Håndtere uventede ting Tilegne sig ny viden

Det synes essentielt at arkitekten har gode kommunikationsevner, da projektets succes er afhængig af om denne får kommunikeret design, ideer, metoder mv. til teamet. Da der arbejdes med et nyt produkt i et ukendt domæne er det også vigtigt at arkitekten evner at tilegne sig ny viden og uventede / ukendte faktorer.

Følgende er en liste af viden en arkitekt bør være i besiddelse af:

Generelt område	Specifikt område
Softwarespecifik viden	Viden om arkitektur koncepter Viden om udvikling af software Viden om software design Viden om programmering
Viden om teknologier og platforme	Specifikke teknologer og platforme Generel viden
Viden om organisationen og ledelsen	Domæneviden Viden om industrien Ledelse og leder erfaring

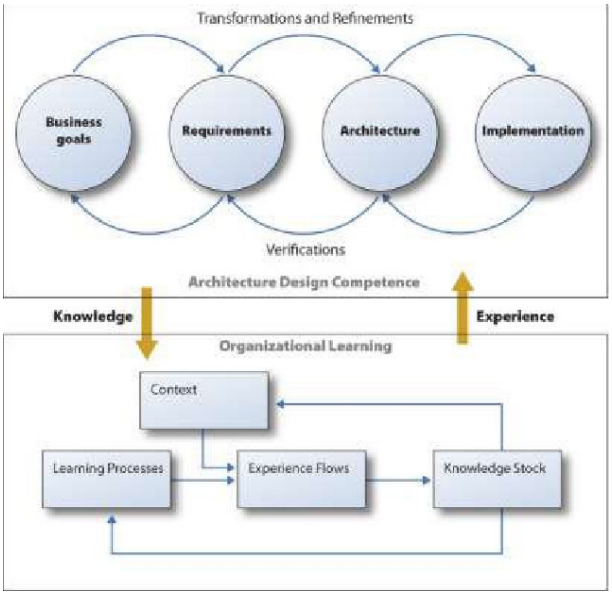
Da denne arkitekt er ”alene” i en organisation uden anden softwareudvikling, er det essentielt at arkitekten besidder den nødvendige softwarespecifikke viden og viden om teknologier og platforme, da det ikke kan forventes at der kan findes hjælp andre steder i organisationen.

“Give examples of how Organizational Coordination and Organizational Learning could ideally take place in this organization.”:

“Organizational Coordination” omhandler en organisations evne til at koordinere akitekturinformation og viden, herunder især hvordan flere afdelinger/teams får koordineret og ideelt set følger en fælles strategi. Da der er gået ud fra det dette er organisationens første softwareprojekt og der kun er dette ene projekt er der ikke umiddelbart flere teams med behov for koordinering. Hvis man forestiller sig at dette produkt skal danne basis for en produktlinie og evt. andre typer produkter, vil der være behov for flere teams og dermed langt større behov for koordinering. I dette tilfælde ville det være oplagt at uddrage erfaringer fra dette ”pilotprojekt”, herunder designmodellen, strategier mv. og gøre denne erfaring til udgangspunkt for andre teams / projekter. Dette kunne fx. gøres ved at nedsætte en basisgruppe med ansvar for at vedligeholde metoder og strategier på tværs af organisationen og samle arkitektonisk viden.

Det essentielle i denne proces er at alle teams benytter de samme metodikker og en fælles base, da koordinering så vil foregå i et ”fælles sprog”.

”Organizational Learning” omhandler en organisations evne til at lære, hvilket vil sige omdanne erfaringer til viden.



Med udgangspunkt i ovenstående figur fra [Bass et al., 2008b] giver viden erfaring, som igen bidrager til en organisations arkitektoniske kompetence. At omdanne viden til erfaring kræver refleksion. At få organisationen til at opbygge arkitektonisk kompetence kræver mao. at refleksion og den resulterende læring er en del af en defineret proces. Foregår dette kun inden for det enkelte team får dette team større arkitektonisk kompetence (ved at reflektere over den opnåede læring). Ved at foretage refleksion på tværs af organisationen opnås vidensdeling og dermed øget arkitektonisk kompetence for hele organisationen.

For at sikre vidensdeling bør refleksion planlægges som en fast del af arkitekturprocessen, dvs. at der i organisationen findes et framework til dette.

“Give examples of what would make the organization architecturally competent.”:

En arkitektonisk kompetent organisation vil have processer på plads til at understøtte koordinering og læring som beskrevet ovenfor. Jo mere processerne er formaliseret og jo mere aktiv fokus de har jo mere kompetent er organisationen. Mere specifikt tænkes det at arkitektur skal drives af den enlige arkitekt, hvorfor meget afhænger af denne persons viden og evner. I den første fase må organisationens samlede arkitektoniske kompetence være lig arkitektens.

“Give examples of what would make the organization architecturally incompetent.”:

Analogt til ovenstående svar så vil en inkompetent organisation ikke være i stand til at understøtte, koordinering, læring og vidensdeling, og igen da dette er organisationens første softwareprojekt er der ingen arkitektonisk viden i organisationen ud over den som arkitekten besidder (eller er i stand til at tilegne sig ved egen hjælp). Det er derfor essentielt at arkitektens viden er tilstrækkelig til at gennemføre de foreslåede aktiviteter og at denne har fornøden viden om domænet.

2.4 Architectural decisions

For the development of the Book Swap application, it has been decided to

1. use Java and RMI as a distributed platform,
2. to use the software architecture methods of Bass et al. (2003),
3. to always program towards an interface, and
4. there cannot be any node that is a server or a client.

Please consider the following:

- What kind of design decisions are the four decisions?
- Using one of the decisions as an illustration, use the template of Tyree and Akerman (2005) to document it (as far as possible)
- Discuss how representations of design decisions could be used in architectural evaluation
- Explain how design decisions could fit into the IEEE 1471 framework

"What kind of design decisions are the four decisions?":

Afgørelsen om at bruge Java og RMI er sandsynligvis taget pga. virksomheden i forvejen anvender Java, og dens medarbejdere har kompetence til at udvikle i Java. Derfor er denne beslutning en Excecutive decision.

Det er besluttet at bruge metoderne fra Bass et al. Denne beslutning er igen en beslutning som må forventes bygger på at arkitekten har gode erfaringer med denne metode. Med andre ord er det tvivlsomt om man ville nå væsentligt andre kvaliteter hvis man havde brugt andre metoder. Denne beslutning er også en Excecutive decision.

Afgørelsen om at altid programmere mod et interface er en (structural) Existence decision, hvor det er fastlagt at man altid skal følge et bestemt programmeringsmønster.

Den sidste afgørelse om at der ikke må være nogen nodes som enten er server eller client, er en Non-existence afgørelse, som er det omvendte af en Existence decision. Denne afgørelse kunne altså omskrives til følgende Existence afgørelse: "alle nodes skal være både klient og server".

"Using one of the decisions as an illustration, use the template of Tyree and Akerman (2005) to document it (as far as possible)":

Issue	Omkostningerne ved ændringer i et system kan let blive store.
Decision	Der skal altid programmeres op mod et interface, hvor det er muligt.
Status	Vedtaget.
Grouping	Modifiability
Assumptions	Det må ikke tage væsentligt længere tid at programmere op mod interfaces i forhold til direkte mod objekterne.
Constraints	Ingen
Positions	Programmere direkte op mod klasser.
Argument	Kunne undlade at bruge interfaces, og programmere direkte op mod klasser, men ved at bruge interfaces får vi en løsere kobling mellem klasserne.
Implications	Det kan have en inflydelse på COTS produkter og eventuelle frameworks hvis disse benytter sig af reflection.
Ralated decisions	Ingen
Related requirements	Ingen
Related artifacts	Ingen
Related principles	Høj modificerbarhed uden stort performancetab.
Notes	Ingen

"Discuss how representations of design decisions could be used in architectural evaluation":

En beskrivelse af arkitekturen og en beskrivelse af arkitektoniske beslutninger er begge en del af den arkitektoniske viden som eksisterer i en eller anden form når et system bliver konstrueret. Arkitekturen lever i systemet, og derfra kan man, i hvert fald delvist, altid genskabe beskrivelsen af den. Men hvis de arkitektoniske beslutninger ikke bliver dokumenteret vil de med tiden gå tabt i takt med at arkitekten glemmer detaljer eller forlader virksomheden. Dermed går vigtig og værdifuld information tabt.

Dokumentation af beslutninger har ikke kun værdi for eftertiden. Den kan også være en hjælp til kommunikationen mellem forskellige aktører. I en evaluering af et systems arkitektur er der af gode grunde flere andre aktører end softwarearkitekten involveret. For at disse skal få indblik i systemets arkitektur er det nødvendigt med en arkitektonisk beskrivelse, f.eks. i views. Views er udmærkede til at beskrive den valgte arkitektur men viser ikke hvad arkitekten har valgt fra, og hvilke overvejelser arkitekten har haft i processen.

Så ved også at beskrive bevidste fravalg, antagelser og kompromiser, som arkitekten på baggrund af grundig analyse har lavet, kan de andre deltagere i en arkitektonisk evaluering få vigtig indsigt arkitekuren

som skal evalueres. Dette er oplysninger som ellers ikke ville indgå i views eller et rationale for arkitekturen. Dermed er det relevant at bruge beslutningerne som input for evalueringen.

Man kunne også holde resultatet af en evaluering op mod de valg der er lavet. Det er muligt at arkitekten bevidst og velovervejet har fravalgt visse løsninger selv om disse virker logiske i en arkitektonisk evaluering.

Man kan uden tvivl drage fordel af at have arkitektoniske beslutninger dokumenterede når en evaluering skal laves. På den anden side kan det måske være en ulempe hvis personerne som skal lave evalueringen kender alle detaljer i baggrunden for valg af arkitektoniske elementer idet at det kan hæmme kritiske spørgsmål. Det er altså vigtigt også at evaluere afgørelserne.

"Explain how design decisions could fit into the IEEE 1471 framework":

Som mange andre metoder og frameworks til at beskrive softwarearkitektur, fokuserer IEEE 1471 på at beskrive arkitekturen som et produkt, og ignorerer eller nedprioriterer væsentligt at dokumentere processen som har ledt til arkitekturen.

IEEE 1471 frameworket foreskriver at man beskriver rationalet for en valgt arkitektur og at man herunder beskriver alternative arkitektoniske koncepter.

Vi ser det dog som at disse beskrivelser er på et højere niveau end mange af de arkitektoniske afgørelser man måtte beskrive efter [Tyree & Akerman, 2005]'s model. Arkitektoniske beslutninger er ofte på et mere detaljeret niveau, end det man typisk beskriver i et rationale.

Dog er et af formålene med IEEE 1471 at dokumentere "the system and its evolution", som det hedder i den indledende tekst til frameworkets beskrivelse [IEEE Computer Society, 2000]. Derfor kunne det være relevant at udvide IEEE 1471 med struktureret beskrivelse af arkitektur beslutninger. Dette kunne gøres som en udvidelse til beskrivelsen af rationale for arkitekturen.

2.5 Architectural Reconstruction

Consider the following (part of a) prototype of Book Swap that the architect of Book Swap has built (where Peer extends Remote):

```
package dk.saip

public class Peer {

            private Collection <String> books = new LinkedList<String>();
            private Collection <Peer> peers = new LinkedList<Peer>();
            private Network network = new Network ();

            public int download (String keyword, Peerto) {
                        for(String book:books) {
                                    if (book. contains (keyword)) {
                                                return network.send(book ,to);
                                    }
                        }
                        return network.send(null , to);

            }

            public void upload (String book , Peer from) {
                        books.add(book);

            }

. . .

}
```

He has decided to implement the network communication in C++ allowing for higher speed.
Please consider the following:

- Explain the Symphony process and how it would apply to this case. No formal architectural documentation exists currently
- What kinds of source views, target view, and hypothetical views would there be in this case? How could they be obtained?
- What would a typical UML tool reverse engineer the prototype into? What would be missing?
- What information would be needed to reconstruct requirements or design decisions?

“Explain the Symphony process and how it would apply to this case. No formal architectural documentation exists currently”:

Symphony går ud på at rekonstruere arkitekturdokumentationen for et allerede udviklet system. Det er generelt en svær proces, da man mister meget information, når man går fra arkitektur til implementation. Symphony er en proces der hjælper til at trække denne information ud af systemet og konvertere det til en arkitekturbeskrivelse.

Overordnet går Symphony ud på at skabe en række source views ud fra systemet. De kan genereres ud fra kildekode, analyser af et kørende system eller opsamlet data efter en kørsel. Source views betragtes normalt ikke som arkitektoniske views, da de indeholder alt for detaljerede informationer.

Ud fra source views skal der laves en mapping til target views. Target views beskriver arkitekturen som den er implmenteret og indeholder den information der skal bruges for at løse problemet.

Som supplement til target views kan hypotetiske views bruges til at beskrive arkitekturen ud fra den information der ellers kan samles om systemet. De hypotetiske views behøver ikke vise den implementerede arkitektur korrekt, men kan bruges til at sammenligne med target views.

Symphony består af to dele, Reconstruction Design og Reconstruction Execution.

Reconstruction Design

Denne del af processen går ud på at finde ud af hvorfor der skal laves en rekonstruktion af systemet og hvilke problemer der skal afdækkes med arkitekturbeskrivelsen, da dette vil have indflydelse på valget af viewpoints. Det kræver mange stakeholders at lave en arkitekturrekonstruktion, så der skal være gode grunde for at udføre den. Disse kan fx være: performance problemer, problemer med vedligeholdelse, ustabilitet, udvidelse eller udskiftning af systemet.

Problemet skal herefter analyseres for at identificere hvilke viewpoints der skal bruges til target views for at give den information der er nødvendig for at løse problemet. Ud fra de valgte target viewpoints skal man finde ud af hvilke informationer der er nødvendige for at generere dem. Source viewpoints beskriver disse informationer, og hvordan man udleder dem fra fx kildekode. For at kunne transformere den opsamlede data i source views skal der defineres nogle regler for hvordan source views kan mappes til target views.

Der kan evt. også defineres nogle hypotetiske viewspoints og deres rolle. Hypotetiske views beskriver ikke nødvendigvis arkitekturen korrekt, men nærmere som den er blevet designet. De kan defineres ud fra evt. eksisterende dokumentation, præsentationer, interviews med stakeholders etc. De hypotetiske views kan fx bruges til at sammenligne med target views, for at vurdere om man er kommet frem til det rigtige resultat.

Der findes ikke nogen formel dokumentation til Book Swap prototypen hvor netværkskommunikationen er blevet implementeret i C++. For at give en bred beskrivelse af arkitekturen vil det være relevant med de tre meget brugte viewpoints; Module, Component & Connector og Allocation, hvor der specielt lægges vægt på kommunikationsdelen. Module view skal identificere hvilke moduler og klasser der benytter netværk, Component & Connector view skal vise hvordan netværket benyttes og allocation view hvorledes de forskellige komponenter er distribueret

Reconstruction Execution

Udførelsen af rekonstruktionen går først og fremmest ud på at trække data ud fra fx kildekode, log-filer, test-kørsler af systemet etc. Der kan fx laves forskellige leksikalske og syntaktiske analyser af kildekode for at finde konkrete informationer om kodens opbygning.

Når source views er lavet skal de konverteres til target views. Der findes desværre stort set ingen gode automatisker måder at konvertere fra source views til target views, så man må ofte gøre det manuelt eller bruge halv-automatiske værktøjer der skal konfigureres til det konkrete tilfælde, så de kan lave den rigtige mapning. De halv-automatiske metoder kan dog nogen gange gentages når de først er blevet konfigureret. Det giver mulighed for at lave rekonstruktion af arkitekturen løbende, når først den er udført én gang.

Reconstruction design og udførelse udføres ofte i iterationer, da der ofte opdages nye muligheder for at trække informationer ud af kildedata under udførelsen og dermed generere bedre target views.

For at lave de source views der er defineret for Book Swap kan man fx udtrække statiske informationer fra de fysiske filer og ved manuel kodeanalyse. En runtime analyse kan laves med en java debugger/profiler til at lave trace, og man kan fx benytte en netværks-sniffer til at undersøge brugen af netværk for eks. chatty kommunikation.

Når target views er genereret ud fra source views skal de analyseres for at afdække det problem der har givet anledning til rekonstruktionen. En repræsentation af arkitekturen skal fremlægges for stakeholders, og der kan fx sammenlignes med en arkitekturbeskrivelse for den version hvor der er brugt RMI, for at vurdere hvad indflydelse det har at skifte netværkskomponenten ud.

”What kinds of source views, target view, and hypothetical views would there be in this case? How could they be obtained?”:

De valgte views er også beskrevet som en del af besvarelsen på ovenstående spørgsmål. De er opsummeret i de følgende tabeller:

Source views:

Source view	Data
Klassediagrammer	Leksikalsk analyse af kode. Brug af Magic Draw.
Sekvensdiagrammer	Trace af output fra kørende system vha. debugger, profiler eller JSeq.
Netværksbrug	Opsamling af netværkskommunikation vha. sniffer, eller ved at bruge DiscoTect med probes der opsamler system events om brug af netværket.
Allokering af moduler	Undersøgelse af build-filer. Der findes ingen gode måder til at lave allokation views på.

Target views:

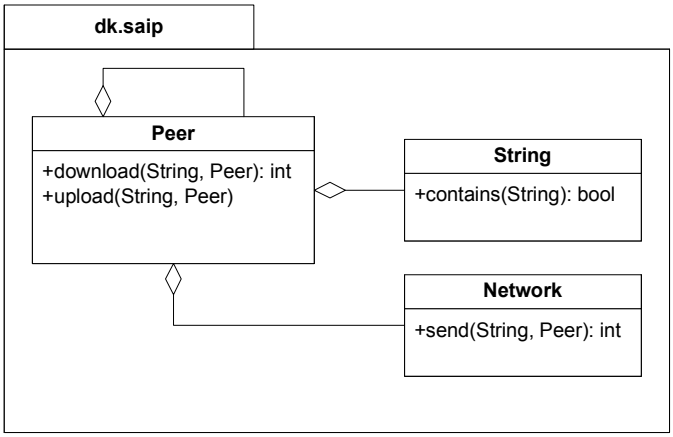
Target view	Data
Module view	Kan laves ud fra de klassediagrammer som fx Magic Draw kan generere. Der vil være noget manuelt arbejde for at få de sidste detaljer på plads.
Component & Connector view	Et værktøj som JSeq kan generere sekvensdiagrammer direkte ud fra en kørsel af systemet. Output fra en netværks-sniffer skal konverteres manuelt til sekvensdiagrammer der beskriver brugen af netværket.
Allocation view	Allocation view må laves manuelt. Man kan være heldig af finde information om konfiguration og allokering af moduler i en build-fil.

Hypotetiske views:

Hypotetisk view	Data
C&C view af Book Swap med RMI	Kan laves som for Book Swap med C++ netværkskomponent, eller eksisterer måske allerede. Dette kan bruges til sammenligning.
Stilarter/taktikker	Ud fra interviews med stakeholders kan der laves views over de brugte stilarter/taktikker. Dette view kan bruges til at vurdere om rekonstruerede arkitekturbeskrivelse er korrekt.
Præsentationer	Præsentationer der er genereret i forbindelse med tidligere reviews.

“What would a typical UML tool reverse engineer the prototype into? What would be missing?”:

Et typisk UML værktøj vil generere UML ud fra ovenstående kildekode som ser nogenlunde sådan ud:



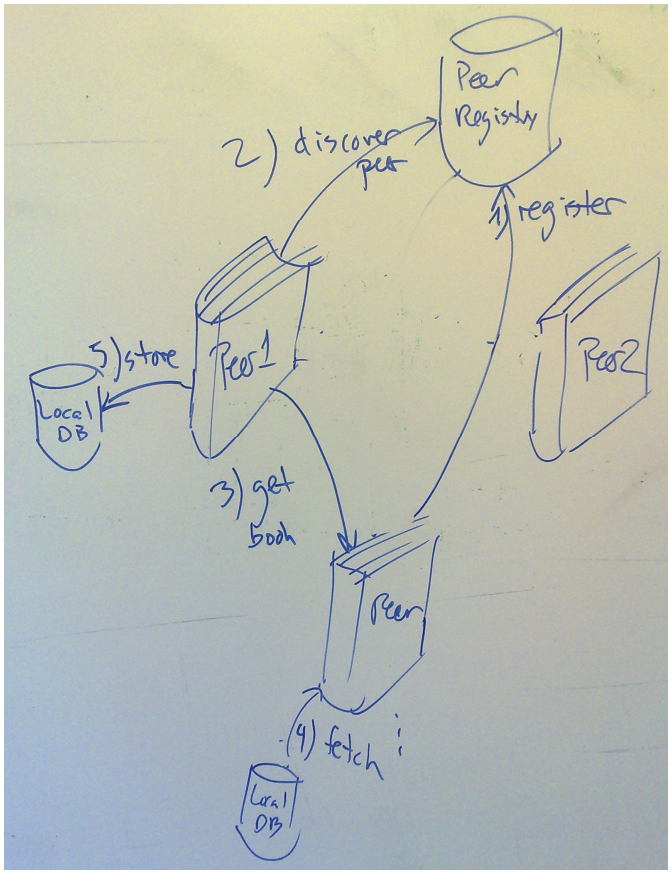
Værktøjer der kan generere UML automatisk ud fra kildekode kan ikke altid se de afhængigheder der er. Hvis fx der findes flere typer af Peers kan værktøjet ikke se hvilke typer der er relevant i hvilke sammenhænge. Værktøjet kan heller ikke se, at listen af strenge rent faktisk repræsenterer bøger. Specielt analyse af C++ kode er svært, da der kan benyttes pointere som kan gøre det stort set umuligt at se hvilke afhængigheder der er ved en statisk analyse.

“What information would be needed to reconstruct requirements or design decisions?”:

Da der ikke findes nogen formel dokumentation af Book Swap prototypen er det nødvendigt at opsamle informationer om krav og beslutninger ud fra interviews med stakeholders, og undersøgelser af fx repositories over tid for at se udviklingen. Det kan også være at der er opsamlet uformelle informationer på en wiki, der kan bruges til at få et overblik over forløbet.

2.6 Connectors

The architect of Book Swap has drawn the following rich picture of his intended architecture:



Please consider the following:

- The architect wants to describe his design using connector types. Outline which types he has chosen
- Which other connector types are/are not applicable to Book Swap?
- The architect decides to implement the system in ArchJava. Discuss the benefits and liabilities of this choice
- Sketch the code of Book Swap in ArchJava (it does not have to compile)
- What consequences does the choice of using ArchJava have for architecture reconstruction?

"The architect wants to describe his design using connector types. Outline which types he has chosen":

De fleste connectorer kan beskrives med forskellige typer, men vi mener at arkitekten har forsøgt at beskrive de følgende typer i sin figur.

Connector	Type
1. Register	Registreringsprocessen fungerer som et procedurekald med parameteroverførsel.
2. Discover Peer	Discover Peer foretages som data access i det globale registry.
3. Get book	Selve funktionen Get book kan beskrives som et procedurekald med en parameter der angiver hvilken bog der skal returneres.
4. Fetch	Fetch kan beskrives som data access i persistent lager.
5. Store	Store er på samme måde som Fetch data access i persistent lager, der opbevarer de bøger der er tilgængelige direkte på en Peer.

"Which other connector types are/are not applicable to Book Swap?":
Afhængig af hvilket niveau man systemet på kan man beskrive forbindelserne mellem Peers og Registry med forskellige connector-typer.

Hvis registreringer bliver publiceret til alle online Peers, kan dette beskrives som en connector med typen Event, der fortæller om en Peer er gået online eller offline.

Hvis man ser selve Registry som en connector kan denne beskrives som en Linkage-type der bruges til at binde Peers sammen run-time.

Hvis selve overførslen af bogen fungerer ved at der oprettes en separat forbindelse på baggrund af en forespørgsel på en bog, kan denne være en stream. For at fordele bøger rundt i P2P-netværket kan der laves en stream med multiple modtagere.

I et P2P-netværk kan man vælge at lave en load-balancering ved at distribuere elementer ud på flere Peers på forkant, for at fordele downloads og dermed øge båndbredden. Dette kan ses som en arbitrator-connector mellem Peers.

Systemet er designet til at fungere på tværs af vidt forskellige Peers, og der er derfor ikke behov for nogen adaptor-connectors.

P2P-systemet indeholder ikke nogen Distributor-connecterer, al kommunikation foregår direkte mellem Peers.

“The architect decides to implement the system in ArchJava. Discuss the benefits and liabilities of this choice”:

Når components og connectors er identificeret i det arkitektoniske design, kan de implementeres i ArchJava. ArchJava vil påtvinge kommunikationsvejene som angivet i koden, og dermed sikre at det arkitektoniske design overholdes. Hvis der er problemer med det arkitektoniske design der gør at forbindelserne ikke kan overholdes, vil ArchJava identificere dem. Den måde ArchJava deler systemet op i komponenter og connectorer på gør det desuden nemmere at skifte en forbindelse ud mellem to komponenter, fx. ændre kommunikationen mellem Peers fra RMI til et dedikeret netværkslag der er designet til overførsel af bøger.

Problemerne ved at benytte ArchJava er de samme som der er beskrevet under afsnit 2.1 Architectural Prototyping.

“Sketch the code of Book Swap in ArchJava (it does not have to compile)”:

Se C2Bravo.zip for et eksempel på hvordan BookSwap kan se ud i ArchJava. Der er lagt vægt på beskrivelsen af connectorer i forhold til ovenstående figur. Dvs. der er ikke taget stilling til hvilken forbindelse der er mellem komponenter.

“What consequences does the choice of using ArchJava have for architecture reconstruction?”:

De statiske component & connector views kan ses direkte i koden, og kan derfor rekonstrueres nøjagtigt umiddelbart ved at se på kildekode. Aktive objekter og de dynamiske egenskaber vil ArchJava ikke give nogen hjælp til.

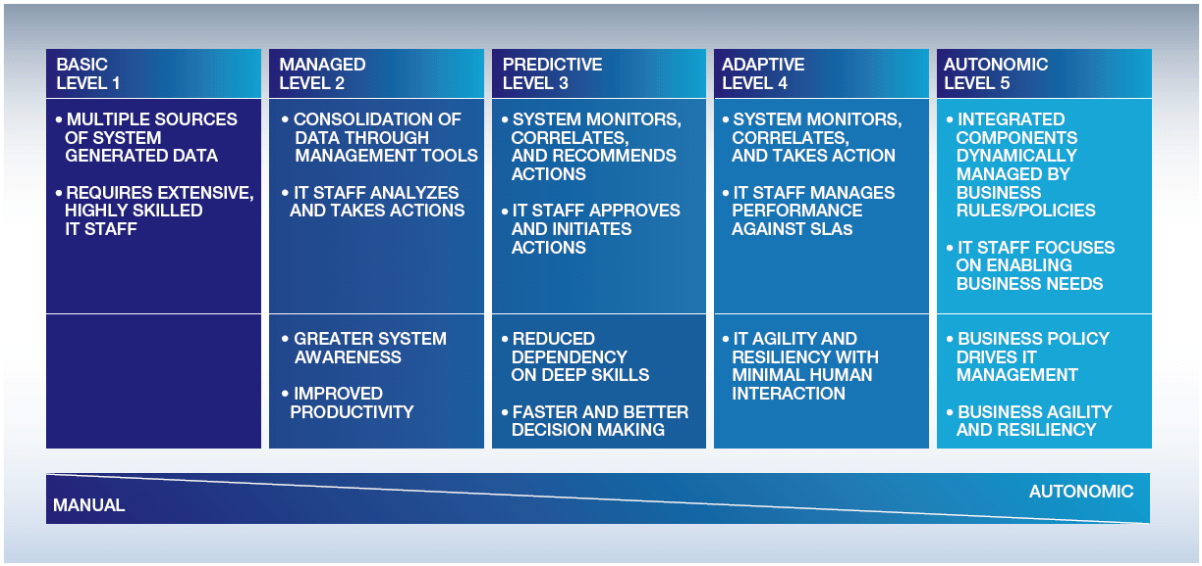
2.7 Architectural reflection

"The customer of the Book Swap system requires that the system becomes self-managed. One specific requirement is that for the mobile version of Book Swap, the system should always choose the best available network connection, trading off cost and bandwidth of the connection. Another requirement is that if one Book Peer fails, it should be discovered and if possible repaired."

Please consider the following:

- Where does Book Swap (with the added requirements) belong on the Autonomic Computing axis?
- Illustrate how the three-layer architecture of Kramer and Magee (2007) could map to Book Swap
- The architect decides to using DiscoTect in the lower layer of the three-layer architecture. Which kinds of probes would he need? Give examples (verbally) of rules that he could write to transform system events into architectural events

"Where does Book Swap (with the added requirements) belong on the Autonomic Computing axis?"



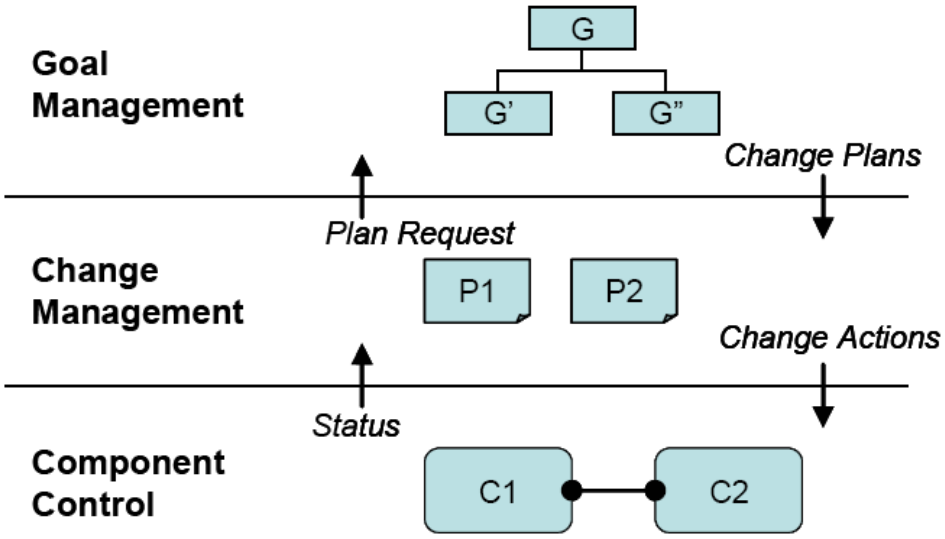
På ovenstående figur ses en specification af graden af autonomi for forskellige computersystemer. Det nuværende system er level 1, da der ikke er systemer eller management tools til at håndtere fejl eller ændrede betingelser.

Foretages de foreslåede ændringer kan (den del af) systemet som minimum betragtes som Adaptive, da det er systemet selv som handler ved at vælge et passende netværk ud fra prisinformation og samtidigt selv reagere "intelligent" hvis en peer fejler.

Der kan argumenteres for at dele af systemet befinder sig på level 5, Autonomic da det er i stand til ud fra Business regler (definitioner af hvad billigste netværk er) selv at vælge det økonomisk mest fordelagtige netværk.

Da det kun er dele af systemet som gøres autonomt kan man diskutere hvor på skalaen systemet som helhed befinder sig.

"Illustrate how the three-layer architecture of Kramer and Magee (2007) could map to Book Swap":



Ovenfor ses [Kramer and Magee, 2007] beskrivelse af trelags arkitektur for autonome systemer

Goal Management

I dette lag håndteres de overordnede mål (hvad man gerne vil opnå). Ud fra disse mål laves en plan som *change management* er i stand til følge. Fejler planen skal *goal management* være i stand til at levere en ny revurderet plan.

I det aktuelle eksempel kunne et mål være at lave en plan for hvorledes netværk skal prioriteres i relation til båndbredde, priser mv.

Evt. kunne *goal management* også stå for en plan for hvordan en fejlende Peer skal håndteres.

Change management

Dette lag har til opgave at fore planen ud I livet og evt. bede om en ny, hvis planen fejler.

Eksempel på dette kunne være at loope gennem Wifi netværk og GPRS netværk i nævnte rækkefølge og så vælge ud fra detekteret båndbredde, samtidigt med der tages hensyn til prisinformation modtaget i planen.

Change management instruerer *component control* i at vælge et bestemt netværk, liste tilgængelige netværk mv.

Component Control

Dette lags opgave er at udføre de faktiske opgaver, hvilket i dette tilfælde vil sige liste netværk, oprette forbindelse til specifikt netværk og evt. måle hastigheder og latency (status) til brug for *change management* for valg af bedste netværk

"The architect decides to using DiscoTect in the lower layer of the three-layer architecture. Which kinds of probes would he need? Give examples (verbally) of rules that he could write to transform system events into architectural events":

Kandidater til prober kunne være følgende:

- Netværkshastighedsprobe
- Netværkstype probe
- Heart beat probe

Alle disse prober benyttes til at måle performance egenskaber for specifikke netværk. Der er i denne besvarelse gået ud fra at prisinformation findes i *goal management* men var det muligt at forespørge på prisen direkte i netværket ville denne information også være relevant, især hvis prisen ikke er statisk, men fx. afhang af tid på dagen, netværksbelastning mv.

DiscoTect er et program til at instrumentere eksisterende kode og ud fra disse events med regler at danne akitekturelle events.

I det ovenstående eksempel er det interessant at undersøge om systemet er i stand til at vælge det korrekte netværk, så følgende er interessant:

- Netværkshastighed falder under en bestemt grænse
- Netværk lokaliseres (typen logges)
- Heartbeat forsvinder fra netværk
- Forbindelse til netværk oprettes
- forbindelse til netværk nedlægges

Ovenstående events vil kunne bruges til at se om der vælges det rigtige netværk og om der vælges et andet hvis det forsvinder eller bliver for langsomt.

Evt. kunne der opsættes regler som detekterer at hastigheden er for lav, kombinere med nedlægning af forbindelse til netværk og oprettelse af et nyt, dvs. lave en kombinationsregel "skift til andet netværk pga. lav hastighed".

3 References

[Bass et al., 2008b] – Bass, L., Clements, P., Kazman, R., Klein, M. (2008).
Models for Evaluating and Improving Architecture Competence. Technical Report CMU/SEI-2008-TR-006

[Hofmeister et al., 2007] C. Hofmeister, P. Kruchten, R. Nord, H. Obbink, A. Ran, and P. America.
A general model of software architecture design derived from five industrial approaches. The Journal of Systems & Software, 80(1):106–126, 2007.

[IEEE Computer Society, 2000] IEEE Computer Society (2000). IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. IEEE Std 1471-2000.

[Tyree & Akerman, 2005] Tyree, J. and Akerman, A. (2005). Architecture decisions: demystifying architecture. IEEE Software, 22(2), pp 19-27.

[MacKenzie et al., 2006] MacKenzie, C. M., Laskey, K., McCabe, F., Brown, P. F., and Metz, R. (2006).
Reference Model for Service Oriented Architecture 1.0. Technical Report Committee Specification 1, 2 August 2006, OASIS.