Location, Location, Communication

Invenio Solutions

Elias Andersson
Bastian Fredriksson
Tobias Hindersson
Tom Johansson
Joel Kallin Kilman
Joakim Lindberg
Elias Lousseief
Edvin Lundberg
Marcus Löf
Martin Steier

Version 1.2 (submitted final draft)

2014-04-09

Abstract

I detta dokument beskriver vi detaljerat arkitekturen för vår lösning. Vi beskriver först hur vår lösning passar in i nuvarande externa system och dessutom vilka gränssnitt som existerar mot dessa. Därefter förklarar vi vår systemdesign och hur vi har kommit fram till den. Vi specificerar även alla komponenter, som exempelvis filer och Java-klasser, i vår lösning utifrån en mall. Utifrån arkitekturen utför vi en genomförbarhetsstudie och drar slutsatser ur den. Slutligen ställer vi upp en tabell i vilken krav från URD-1 binds ihop med komponenter.

Table of contents

1 Introduction 1

1.1 Purpose 1

1.2 Scope of the Software 1

1.3 Definitions, acronyms and abbreviations 1

1.4 References 5

1.5 Overview of the document 6

2 System Overview 7

3 System Context 11

3.1 Gränssnitt: Databas 19

3.2 Gränssnitt: Applikationsserver 20

3.3 Gränssnitt: GCM-server 20

4 System Design 21

4.1 Design method 21

4.2 Decomposition description 29

5 Component Description 65

6 Feasibility and Resource Estimates 109

7 User Requirements vs Components Traceability Matrix 117

Document Change Record

Datum	Version	Vad ändrades?
2014-03-24	0.1	• Lade till Kapitel 1, 2, 3 och 4. • Lade till Appendix A, B och C.
2014-03-26	0.2	• Lade till Kapitel 5, 6 och 7.
2014-04-01	0.3	• Lade till Abstract. • Skrev om och utökade i stort sett alla kapitel. • Lade till nya bilder.
2014-04-07	1.0	• Utökade alla kapitel markant med text, bilder och tabeller. • Tog bort Appendix A, B och C. De är nu integrerade i olika kapitel på ett smidigt sätt.
2014-04-08	1.1	• Utökade Kapitel 1.3 och 1.4. • Ändrade numreringen på vissa källor. • Lade till en bild i Kapitel 5.23.9. • Ändrade formateringen i Kapitel 5. • Korrigerade en felaktighet i Kapitel 4.2.2. • Åtgärdade att vissa begrepp förklarades med olika termer i olika kapitel.
2014-04-09	1.2	Korrigerade några mindre felaktigheter i Kapitel 5.4, 5.5 och 5.11.

1 Introduction

För att kunna programmera ansvarsfullt, framtidssäkert och framför allt effektivt, krävs planering och väldefinierade ramar inom vilka programmeraren kan hålla sig inom. Detta dokument kommer göra det möjligt för oss att slutföra projektet inom given tidsram.

1.1 Purpose

Syftet med detta dokument är att vara en mall som vi kan vända oss till när vi programmerar. Målet är att den slutgiltiga produktens arkitektur skall följa detta dokument. Dessutom baseras planeringen i PPD-2 och Kapitel 6 i detta dokument på systemets olika tidsberoenden. Dokumentet är inte lagligt bindande, men det ska krävas goda anledningar för att något ska ändras i den arkitektur som beskrivs, eftersom ändringar fortplantar sig och kan orsaka förseningar och ändringar av komponenter som redan är färdiga. Detta gäller hela den arkitekturella delen av ADD-1.

1.2 Scope of the Software

Produkten är en Android-applikation som ska användas inom sjukhus för vårdpersonal. Applikationen ska kunna hitta specifika människor och förmedla deras position. Applikationen ska kommunicera via en server och databas men inte lagra några positioner.

1.3 Definitions, acronyms and abbreviations

API - Application Programming Interface. Det gränssnitt genom vilket man kan använda en komponent av något slag. Används typiskt för programbibliotek och ibland även för applikationsservrar och liknande.

Beacon – En enhet som sänder ut BLE-signaler.

BID – Beacon ID. Fyra bytes (två bytes minor och två bytes major) som används för att identifiera en beacon.

BLE – Bluetooth low energy. En version av Bluetooth som är designad för att dra mindre ström än vanlig Bluetooth.

Bluetooth - Kan referera till en protokollstack och en typ av hårdvaruchip som är vanlig för att på korta avstånd trådlöst överföra data mellan tekniska apparater. Den senaste standarden är Bluetooth 4.0.

Brute force-attack - Ett försök att komma åt ett lösenord genom att låta en dator systematiskt testa alla möjliga kombinationer av lösenord i en viss serie, som t.ex. alla bokstäver i alfabetet.

Callback - Anrop från en metod till en annan metod där den första metoden förväntar sig att få ett svar tillbaka.

CIF – Clinical Innovation Fellowship. Projektets arbetsgivare. CIF är en del av CTMH (Center for Technology in Medicine and Health), som är ett samarbetsprojekt mellan Karolinska Institutet, Kungliga Tekniska högskolan och Stockholms Läns Landsting.

Connector/J - Specifik drivrutin för kommunikation mellan JDBC och en MySQL-databasserver.

DBMS - Database Management System. Ett komplett databassystem som innehåller möjligheter till att såväl samla och strukturera data som att underhålla och administrera denna.

Debugger - Ett verktyg som används för att hitta fel (buggar) i ett program. Kan bland annat användas för att pausa det program som testas vid i förväg specificerade tidpunkter (break points) under programmets körning för att kunna undersöka att all data i programmet ser ut som förväntat. Kan även användas för att köra koden i programmet som testas en rad åt gången och på så sätt följa vad som händer i programmet steg för steg.

DeviceID - Registerings-ID i form av en textsträng på upp till 4096 byte som GCM använder för att identifiera en Androidtelefon.

EER-diagram - Enhanced entity-relationship (EER) är en modell som används för att designa databaser. EER är en utökning av entity-relationship (ER).

Foreign key - En nyckel i en databastabell som unikt identifierar en rad i en annan tabell.

foreignUID – Användarnamn för den person som blir förfrågad.

GCM – Google Cloud Messaging. En tjänst som Google tillhandahåller för att låta en tredjepartsserver skicka push-meddelanden till en Android-telefon som har registrerat sig på GCM. Google Cloud Messaging är en del av Google Play Services. När en Android-telefon registrerar sig på GCM erhåller den ett registreringsid som skickas till vår tredjepartsserver. Denna server använder detta registreringsid och en API-nyckel till GCM för att skicka push-meddelanden till klienten via GCM över POST. Ett pushmeddelande är ett litet datapaket med information, t ex begäran om att skicka beacon-id. När ett pushmeddelande kommer till GCM-servern slår Google upp rätt IP-adress med hjälp av registreringsid och skickar meddelandet till telefonen. Android-operativsystemet läser meddelandet och startar den applikation som meddelandet hör till. Att använda GCM sparar batteritid eftersom telefonen inte behöver hålla en kommunikationskanal öppen till vår server, dessutom håller GCM koll på klientens IP-adress.

Handle – Är ett abstrakt begrepp. När en anslutning till servern etableras interagerar klienten med servern genom detta handtag.

HTTP - Hypertext Transfer Protocol. Ett kommunikationsprotokoll som används för att överföra webbsidor på World Wide Web.

HTTP POST - En förfrågningsmetod för att begära information över HTTP-protokollet. Används ofta av webbaserade tjänster för att låta användaren ladda upp filer eller skicka information mellan olika sidor. Push-meddelanden som skickas till GCM, skickas med hjälp av POST. I Java kan man skicka data till en webbserver över POST med hjälp av klassen HttpUrlConnection.

Intent - En abstrakt beskrivning av en operation som ska utföras.

I/O - Input/Output. Förkortning för inmatning till och utmatning från en dator. Används som samlingsbegrepp för de medel med vilka en användare kan kommunicera med en dator (knappar, tangentbord, skärm, mus och så vidare).

IP-adress - Den adress som används för att adressera datapaket till enskilda enheter över IP-baserade nätverk.

JDBC - Java Database Connectivity. Javas generella bibliotek för kommunikation med databaser.

JSON - JavaScript Object Notation. Textbaserat standardiserat dataformat som vi använder för att skicka data mellan klient och server. JSON består av nyckel-värde-par separerade med kolon. JSON är ett semistrukturerat format precis som XML, men är enklare att använda.

JSON.simple - Ett bibliotek med öppen källkod för kodning och avkodning av JSON-strängar i Java.

JRE/JDK - Java Runtime Environment (JRE) består av en virtuell maskin som kan köra Javaprogram. Java Development Kit (JDK) innehåller bibliotek och komponenter för att kunna utveckla program för Java.

LAN - Local Access Network, lokalt datornätverk. Typiskt sett fysiskt närbelägna datorer som kopplas samman genom någon form av nätverksenhet såsom en router. Begreppet används som kontrast till WAN (Wide Area Network) vilket motsvarar hela Internet.

Logcat - Verktyg i Eclipse för att se meddelanden från en fysisk Android-telefon eller emulator som körs. Logcat innehåller statusinformation från operativsystemet, programmerarens debugutskrifter och felmeddelanden.

Method chaining - Ett designmönster som används för att kedja ihop flera metodanrop efter varandra för att göra koden mer lättläst. En mer utförlig beskrivning finns i Kapitel 4.1.3.4.

Major – Värde som används för att identifiera en grupp av beacons. Sparas i klassen Beacon som major = 2^8*byte1+byte2.

Minor – Värde som används för att identifiera en specifik beacon i en grupp. Sparas i en klass Beacon som minor = 2^8*byte1+byte2.

MSC – Mobile Switching Center. Den primära delen av en GSM-server som används för att dirigera trafik från en klient till en annan. I vårt fall är det den del av servern som är ansvarig för besvara och skicka ut positionsförfrågningar.

MSD – Manufacturer Specific Data. Serienumret för de beacons vi skall kommunicera med.

MySQL - En av de populäraste DBMS som dessutom är gratis och har öppen källkod. Utvecklas av mjukvaruföretaget Oracle.

Opcode – Operation code. Koden för den operation som ska utföras.

Payload - Den information, i form av data, som skickas med ett metodanrop eller annan dataöverföring.

Proximity UUID – En unik identifierare bestående av ett 128-bitarsvärde som kan identifiera en eller flera beacons.

RSSI – Received signal strength indication. Beskriver hur stor effekt den mottagna radiosignalen har.

SDK - Software Development Kit. Programbibliotek som tillhandahålls för utvecklare.

Serverdemon - En process som körs i bakgrunden, lyssnar efter inkommande nätverksanslutningar och tar hand om de som anslutningar som kommer in till servern.

Skräpsamlare - En process som frigör datorminne som ett program inte längre använder. Behövs för att undvika att datorns arbetsminne blir fullt.

SQL - Structured Query Language. Query-språk som används för interaktion med den vanligaste typen av databaser.

Surrogatnyckel - En automatiskt genererad nyckel i relationsdatabaser som inte har någon egentlig anknytning till resterande attribut i tabellen.

TCP - Transmission Control Protocol, transportlagrets viktigaste protokoll enligt TCP/IP-modellen. Protokollets syfte är i stort att se till att ett datapaket som färdats över ett nätverk ska nå rätt process när det väl hamnat i mottagarens dator.

TCP/IP-modellen - En teoretisk modell över strukturen på datornätverk och användningen av dessa, i form av en protokollstack. Baseras på abstraktion mellan lager där det lägsta lagret är hårdvarunära och det högsta är mjukvarunära.

Token – Genereras av servern när en användare loggar in. Används för att autentisera användaren när förfrågningar skickas till servern.

UI - User Interface. Gränssnittet som användaren använder för att interagera med applikationen.

UUID - Universally Unique Identifier. Ett serienummer som används för att identifiera ägaren till en beacon. Serienumret består av 16 bytes på formatet XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX.

VLR – Visiting Location Register. En databas som innehåller alla användare som är inloggade på en viss tjänst.

XML - Extensible Markup Language. Ett märkspråk som används av Android för att beskriva hur användargränssnittet för olika klasser och fragment ska se ut.

XMPP – Extensible Messaging and Presence Protocol. Ett kommunikationsprotokoll som är baserat på XML.

1.4 References

[1] FluentInterface, Fowler M., http://martinfowler.com/bliki/FluentInterface.html, hämtad 2014-03-21.
Innehåll: Handlar om begreppet fluent interface som myntades av Martin Fowler och Eric Evans. Ett fluent interface är ett API som är skrivet på ett sådant sätt att instansiering av nya objekt liknar naturligt språk.

[2] Mediator Design Pattern, http://sourcemaking.com/design_patterns/mediator, hämtad 2014-03-23.
Innehåll: Innehåller en utförlig förklaring om Mediator-mönstret.

[3] Android Developer Reference, Activity | Android Developer, http://developer.android.com/reference/android/app/Activity.html, hämtad 2014-03-26.
Innehåll: Här kan man läsa om metoder i klassen Activity samt vad Activites använts för i Android. Det finns också digram för livscykeln för en Activity, hur man startar activities hur man kommunicerar mellan dem med mera.

[4] Android Developer Reference, Fragment | Android Developer, http://developer.android.com/reference/android/app/Fragment.html, hämtad 2014-03-28.
Innehåll: Dokumentation för fragments i Android. Samma innehåll som 4.1.2 fast för fragment.

[5] json-simple - JSON.simple - A simple Java toolkit for JSON - Google Project Hosting, http://code.google.com/p/json-simple/, hämtad 2014-03-20.
Innehåll: Google Code-sidan för JSON.simple. Här finns all dokumentation och källkod för projektet.

[6] MySQL 5.1 Reference Manual :: 13 SQL Statement Syntax :: 13.1 Data Definition Statements :: 13.1.17 CREATE TABLE Syntax, https://dev.mysql.com/doc/refman/5.1/en/create-table.html, hämtad 2014-04-06.
Innehåll: Dokumentation för MySQL över hur man skapar nya tabeller och hur syntaxen är uppbyggd.

[7] SharedPreferences,
http://developer.android.com/reference/android/content/SharedPreferences.html, hämtad 2014-04-07.
Innehåll: Javadoc för SharedPreferences.

1.5 Overview of the document

Kapitel 2 beskriver systemet på en överskådlig nivå och kan således användas för en snabb insikt i arkitekturen, medan Kapitel 3 går in på en mer detaljerad nivå. Kapitel 4 beskriver systemets design och vilka resonemang som bidragit till besluten som tagits. Kapitel 5 beskriver varje komponent i detalj. Kapitel 6 behandlar huruvida projektet är genomförbart eller inte, och klassificerar dessutom alla komponenter efter prioritet och need från kraven i URD-1. Kapitel 7 redovisar kopplingen mellan komponenterna och de krav vi ställde upp i URD-1.

2 System Overview

2.1 Externa system och gränssnitt

I detta kapitel beskrivs de externa system och gränssnitt som systemet kommer vara beroende av för att fungera. De flesta av dessa beroenden kommer från att vi har valt att utveckla applikationen för Android.

2.1.1 Google Cloud Messaging (GCM)

Positionsförfrågningar från en användare till en annan användare kommer att skickas via GCM HTTP, en tjänst som tillhandahålls av Google för utvecklare av Android-applikationer. GCM gör det möjligt att skicka push-förfrågningar från en applikationsserver till en användare av en applikation, vilket innebär att användarens Android-enhet kan ta emot och behandla meddelanden även om applikationen bara körs i bakgrunden. Det enda som krävs är att enheten är ansluten till internet och registrerad med GCM. Efter att utvecklaren implementerat GCM i applikationen sköts registreringen utan att användaren behöver göra något.

En förfrågan i GCM HTTP skickar en JSON-sträng över HTTP POST. Svar från klienter kommer däremot inte att skickas via GCM utan skickas direkt till servern. Servern tolkar sedan svaret den fått från den förfrågade klienten, kopplar detta till ett rum och skickar rumsnamnet till den klient som efterfrågat positionen.

2.1.2 Android SDK, beacon SDK, Java

Eftersom applikationen ska utvecklas för Android-enheter så behöver både Javas och Androids SDK användas. För kommunikationen med de beacons som används behövs även dessa beacons SDK. Alla dessa SDK:er innehåller bibliotek med funktioner som måste användas för att skriva applikationen.

2.1.3 SQL

Vi kommer behöva lagra information om användare, rum och beacons för att systemet ska fungera. För att enkelt kunna lagra och distribuera denna information tyckte vi det var naturligt att använda en databas. Vidare tyckte vi att det bästa valet var en SQL-databas eftersom den dels erbjuder god prestanda, dels låter oss bestämma vilken information som ska hanteras och hur den ska lagras. Det finns olika implementationer av SQL men vårt val föll på MySQL eftersom det är en välkänd och väldokumenterad implementation.

2.2 Systemdesign

Den tänkta lösningen innefattar tre huvudkomponenter, nämligen:

- Beacons
- Handhållna Androidenheter med vår applikation installerad
- Ett centralt nav i form av en server med tillhörande databas

2.2.1 Beacons

För positionering kommer beacons att placeras ut på fasta platser, exempelvis i rum. Dessa beacons skickar kontinuerligt ut en signal via Bluetooth. Varje beacon har ett unikt beaconID (BID) som kan användas för att skilja dem åt.

De handhållna enheterna kommer att lagra vilken beacon de befinner sig närmast för tillfället genom att mäta signalstyrkan på alla från beacons mottagna utsändningar och lagra BID tillhörande den beacon med starkast signalstyrka. BID kopplas sedan i databasen till en fysisk plats, exempelvis ett rumsnummer, när en förfrågan görs.

2.2.2 Klientdesign

Klienten ska ha en så enkel design som möjligt för att inte kräva onödigt mycket prestanda, men även för att vara enkel att använda och kräva minimal inlärning. Den slutgiltiga produkten ska underlätta användarnas vardag och det är därför viktigt att användargränssnittet inte hindrar detta genom att vara onödigt komplicerat.

Vi har därför valt en tabbdesign där det mesta går att nå från en enda huvudvy. På så sätt behöver användarna inte leta sig fram igenom flera olika vyer för att hitta en viss funktion. Denna design har tre huvudtabbar: Sökning, favoriter och loggbok. Sökning används för att söka efter användare, favoriter visar en lista över användare som lagts till som favoriter, och loggbok visar en lista över vilka som begärt den inloggade användarens position. I alla dessa listor kan man klicka på en användare för att komma till en vy som visar mer information om den valda användaren. Dessutom ges möjlighet att begära den valda användarens position.

Förutom denna huvudvy finns det även en inloggningsskärm som visas om ingen inloggningsinformation finns sparad när applikationen startas, och dessutom en vy för inställningar.

2.2.3 Databas

Databasen består i huvudsak av information om tre objekt; användare, rum och beacons. Dessa kopplas till varandra genom ett flertal relationer. Exempelvis kopplas beacons till de rum de är monterade i, och användare kopplas till de avdelningar de arbetar för och de arbetsroller som de har. Den valda databasdesignen är utformad för att minimera dubbellagring och därigenom upprätthålla en god prestanda.

Alla som ska använda applikationen måste finnas registrerade i databasen för att kunna logga in. Informationen som registreras i databasen är bland annat användarnamn, lösenord, namn, arbetsroll och eventuella kontaktuppgifter.

För att förhindra att obehöriga får tillgång till tjänsten kommer det inte vara möjligt för användare att själva registrera konton. Detta kommer istället att skötas av en systemadministratör som lägger till nya användare i databasen.

2.2.4 Inloggning

Vid inloggning ska användarnamn och lösenord matas in av användaren i klienten. Inloggningsuppgifterna kommer att skickas tillsammans med opcode och deviceID till servern för verifikation. Vid korrekt angivna uppgifter genererar servern en unik token som skickas ut och denna sparas av klienten. Endast en token är giltig åt gången för varje användare och vid varje inloggning genereras en ny.

2.2.5 Token

Den unikt genererade token som skickats från servern till klienten efter en lyckad inloggning kommer att sparas av klienten. Efter inloggningen kommer klienten att skicka med denna token i meddelanden och förfrågningar till servern för att säkerställa att meddelandena kommer från auktoriserade enheter. Detta syftar till att identifiera användaren utan att behöva skicka användarens inloggningsuppgifter upprepade gånger. Det finns två huvudsyften till detta förfarande, nämligen att hindra användaren från att vara inloggad på flera enheter samtidigt och att minska risken för att obehöriga tillskansar sig användarens inloggningsuppgifter via en så kallad brute force-attack.

Skickar applikationen en ogiltig token kommer servern att svara med en begäran om att applikationen ska logga ut och att användaren tvingas logga in på nytt för att tilldelas en ny token. Sannolikheten att någon ska lyckas generera en fungerande token utan att få den från servern är låg eftersom den är en lång slumpmässigt genererad sträng och den är bara giltig

under en inloggning. Tanken är även att användarens konto ska blockeras efter ett flertal misslyckade inloggningsförsök för att ytterligare öka säkerheten.

Integritet

För att förhindra att användarna missbrukar systemet ska samtliga positionsförfrågningar loggas i servern. Anledningen till detta är att det ska vara möjligt att i efterhand ta reda på om någon utnyttjat systemet på ett otillbörligt sätt. Det ska däremot inte vara möjligt att spåra användare och därför kommer inte användarnas positioner att lagras i databasen. Dessa kommer istället att skickas direkt från den förfrågade användarens telefon via servern till den användare som efterfrågat positionen. Användarnas telefoner kommer bara att lagra senast kända BID och dessa kommer endast att kopplas till en position genom uppslagning i databasen.

3 System Context

Syftet med vårt projekt är att förverkliga och implementera idén om en tjänst som våra projektgivare utvecklat. På en grundläggande nivå är det sedan länge fastställt vilken typ av information som ska förmedlas genom denna tjänst, men eftersom vi nu håller på att gå in i en mer konkret fas av projektet krävs det att vi i mer konkreta ordalag kan formulera den produkt som vi ska bygga.

En rad arkitekturella beslut väntar men ett av de mindre svåra sådana är att vi kommer att skapa vår produkt delvis genom användning av andra, befintliga produkter. Inom programmering pratar man ofta om modularisering vilket innebär att man strävar efter att tydligt avgränsa och gruppera delar och funktioner inom ett givet projekt (se *coupling* och *cohesion* i Kapitel 4.2.3). På en låg nivå kan man med detta mena att man bör samla kod som har liknande syfte och berör samma aspekter av något större på samma ställe. På en hög nivå kan detta dock även innebära att man i praktiken använder befintliga lösningar som finns på marknaden då man som del i ett projekt behöver någon viss egenskap eller funktion som man vet redan existerar. En klar fördel med detta är givetvis att de som utvecklat produkten som man tänkt sig att använda säkerligen har mer erfarenhet av just den funktion som deras produkt fyller, och att de därigenom sannolikt gör ett betydligt bättre jobb på området än vad man skulle göra själv. Vidare bidrar detta även till modulariseringen av det egna projektet vilket också gör att koden blir lättare att underhålla. Dessutom vore det även väldigt onödigt att uppfinna hjulet om och om igen; genom att använda kunskap från andra och lägga till egna delar bidrar vi till vår globala samlade kunskap, och kanske kan vår produkt i framtiden utgöra resurs i något ännu större projekt?

I praktiken innebär alltså mjukvaruutveckling, på såväl hög som låg nivå, till stor del att kombinera och anpassa existerande lösningar och funktioner till något som i slutändan har förmågan att fylla den funktion som man har som mål att fylla med sitt projekt.

Vårt projekt kommer alltså under implementationen i de flesta fall att nyttja såväl existerande hårdvara som mjukvara. De resurser som den i huvudsak egenskrivna delen av projektet omger sig med[1], kommunicerar via och delegerar funktionsansvar till kallas för dess kontext och den

[1] Visserligen använder även den egenskrivna delen av projektet fördefinierade bibliotek med funktioner, men dessa byggstenar, som i förväg är tillgängliga, är i sammanhanget så små att delarna ändå kan betecknas som egenskrivna.

kan alltså begrundas ur såväl högnivåperspektiv som lågnivåperspektiv, med avseende på både mjukvara och hårdvara. Ett systems kontext är beroende av var man väljer att dra gränsen för det aktuella system man studerar. För att kommunikationen med dessa externa resurser ska vara möjlig sker denna kommunikation alltid enligt någon specifik form vilken dock kan skilja sig avsevärt från fall till fall. Man säger att en resurs kan utnyttjas genom dess gränssnitt (eng. *interface*). Använder man programvara från annan part tvingas man i allmänhet rätta sig efter deras fördefinierade gränssnitt medan man, om man själv konstruerar två komponenter som ska kommunicera, kan utforma gränssnittet till en av dessa på egen hand. I allmänhet får den som initierar kontakten med annan mjukvara rätta sig efter dess gränssnitt och man kan kort och gott se det som kontaktytan utåt som möjliggör användande av produkten i fråga.

Den egenskrivna programvaran som kommer att utgöra stommen i detta projekt kommer alltså att använda sig av extern såväl hårdvara som mjukvara. Vi kommer dock under denna punkt att koncentrera oss på systemkontext ur mjukvarusynpunkt med begreppet system konkretiserat som all den programvara som utgör vår mjukvarulösning, och då med särskilt fokus på den osynliga gräns som finns mellan egenskrivna delar och övriga, externa delar. Vi kommer alltså inte ägna rum åt specifikationer av hårdvara, delvis med anledning av att denna, bortsett från de beacons som kommer att användas, är relativt flexibel i sitt utförande i jämförelse med mjukvaran. Projektet är i allra högsta grad ett mjukvaruprojekt och det är på denna abstraktionsnivå som produktens innovationspotential ligger.

Tre typer av mjukvarugränssnitt

Vi ska nu börja med att definiera och skilja mellan några typer av gränssnitt som på olika nivåer alla är viktiga för produktens slutresultat. Härvid bör påtalas att ordet “komponent” typiskt sett är ett begrepp som används för att åsyfta något i stil med “mindre del i något större”, och där ordets konkreta betydelse kan komma att skifta beroende på vilken typ av helhet man för ögonblicket avser. Typiskt sett kommer vi dock att betrakta vårt projekt på mjukvarusystemnivå (hädanefter även kallat systemnivå) där alltså server, klient och så vidare kan betraktas som komponenter varför vi använder ordet komponent som ett relativt ord och systemkomponent (egentligen mjukvarusystemkomponent) som benämning på de konkreta komponenter som finns på mjukvarusystemnivå.

Det sociala gränssnittet är det som utgörs av telefonens I/O-enheter med vilka användaren kan bruka produkten. Detta perspektiv kallar vi för interaktionsperspektivet och i allmänhet faller ansvar för utförandet av detta gränssnitt exempelvis på en så kallad interaktionsdesigner.

Betraktar vi ett slutet mjukvarusystem finns även olika gränssnitt mellan de i lösningen ingående komponenterna och vi kallar detta mjukvarusystemperspektivet. En än mer subtilare typ av gränssnitt finner vi mellan egenskrivna klasser och officiellt tillhandahållna Javaklasser eller andra bibliotek som skrivits av mjukvaruleverantörer. I detta perspektiv betraktar vi som ett generellt system specifikt en mjukvarusystemkomponent och vi kallar alltså perspektivet för mjukvarusystemkomponentperspektivet.

Egentligen beskriver dessa tre gränssnitt samma typ av funktion, nämligen kommunikationen mellan två komponenter i ett system då den ena komponenten vill använda sig utav den andra. Distinktionen som gjorts dessa emellan härrör endast ur vilken abstraktionsnivå man haft vid definitionen av de relativa begreppen system och komponent; i det sociala gränssnittet består systemet (“helheten”) av två komponenter vilka är mjukvarusystemet och användaren, medan man i det andra fallet ser mjukvarusystemet som en helhet och därför definierar dess komponenter efter den största möjliga och meningsfulla uppdelning som går att göra ur detta perspektiv. Det går att undersöka ytterligare lägre och även mer hårdvarurelaterade abstraktionsnivåer med avseende på systemkontext, men då sker undersökningen med utgångspunkten att mjukvarusystemet, alltså programvaran, såväl egenskriven som redan existerande, står för helheten, motsvarande den andra typen i ordningen av gränssnitt återgivna i föregående stycke. Vidare kommer vårt fokus i synnerhet att hamna på gränssnitt mellan det subsystem som utgörs av vår egenskrivna mjukvara och den övriga delen av mjukvarusystemet.

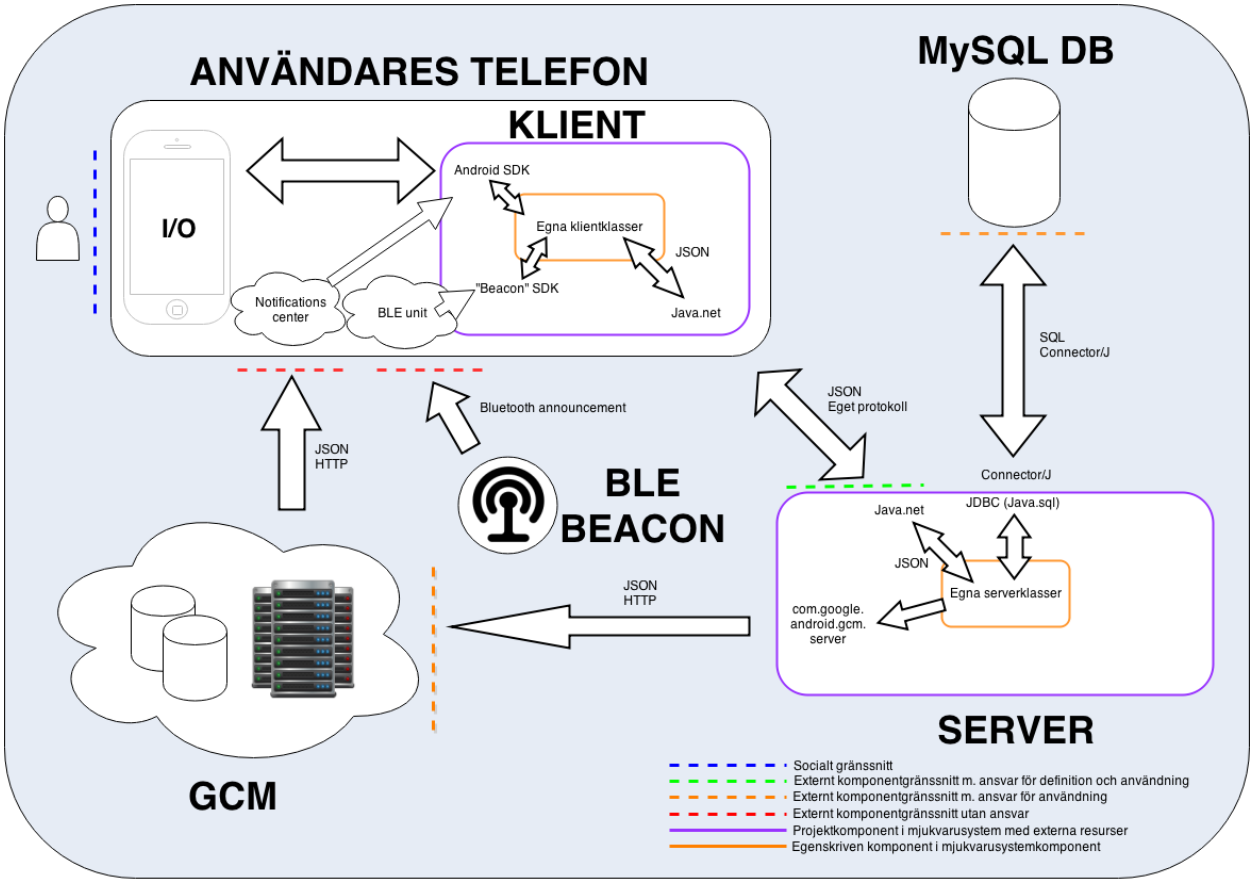
I tabellen nedan beskrivs några exempel på hur konkretiseringen av begreppet komponenter kan se ut beroende på vilket systemperspektiv man använder.

Perspektiv	**Komponenter**
Interaktionsperspektiv	Användare, produkt
Mjukvarusystemperspektiv	Mjukvarusystemkomponenter, server, klient, ...
Mjukvarusystemkomponentperspektiv	Klasser, externa bibliotek, inbyggda bibliotek
Hårdvarusystemperspektiv (LAN)	Datorer, nätverkskablar, routrar
Hårdvarukomponentperspektiv (smartphone)	Skärm, knappar, radioantenn, BLE-enhet, batteri

Klasser och bibliotek i mjukvarusystemkomponentperspektivet

I flödesschemat nedan ser vi egenskrivna systemkomponenter i vår mjukvarulösning inom de heldragna lila linjerna, medan man innanför dessa finner renodlat egen kod inom de heldragna

orange linjerna. Utanför de orange linjerna, men innanför de lila, finner man bibliotek och klasser som använts för att utöka den egna funktionaliteten. Kontaktytan dessa emellan utgörs av gränssnitt på systemkomponentnivå då vår egenskrivna kod importerar kompletta bibliotek (komponenter om man ser en systemkomponent som en helhet i sig själv) och sedan använder dessa genom de publika klasser och metoder som tillhandahålls (dessa tillsammans utgör bibliotekets gränssnitt, även kallat API). I realiteten sker dock övergångar och datautbyte mellan egenskrivna klasser och externa bibliotek sömlöst och inte lika konkret längs ett tänkt vertikalt snitt som i många andra fall. Detta har försökt åskådliggöras i schemat genom att i huvudsak framställa avgränsningar som utgörs av de lila heldragna linjerna som enhetliga och egenproducerade komponenter vilka dock har en mer eller mindre sömlös underdelning i egenskrivna klasser och externa bibliotek.



Ett biblioteks gränssnitt är upprättat helt efter de riktlinjer som den som tillhandahåller biblioteket bestämt. Vi kan använda biblioteket genom att anpassa oss till dessa riktlinjer och använda de namngivna metoder och returvärden som finns definierade i bibliotekets specifikationer och koddokumentation. Källkoden till bibliotek går i allmänhet att ändra på, även om man oftast inte

gör det. Gränssnittet mot dessa kan i vidare bemärkelse, och då det handlar om bibliotek för kommunikation, även ses som en första del i ett gränssnitt mot en annan komponent i systemet. Så är ofta fallet eftersom kommunikation mellan enheter på elektronisk väg typiskt innebär interaktion med hårdvara vilken man gärna kapslar in i sin helhet och använder som separat komponent, eftersom det annars skulle innebära väldigt mycket extra arbete för programmeraren.

Det bör påpekas att endast utomstående bibliotek som har en avgörande roll i produktens övergripande informationsflöde har illustrerats i schemat och att paketresurser av mer trivial art inte tagits med.

Detaljerad förklaring till flödesschema

Gränssnitt mellan systemkomponenter innebär den kontaktyta som finns mellan en anslutande komponent och den komponent som den ansluter till. Dessa går lättast att hitta genom att se till var de avgränsade komponenterna i systemet kommer i kontakt med varandra. I vissa fall är det inte givet var dessa gränser går och man skulle exempelvis kunna se själva klientapplikationens kontaktyta med telefonen som ett gränssnitt. Dock sker denna kommunikation genom sömlös integrering av klasser ur Androids SDK i de egna klientklasserna, vilket i slutändan är att se som ett gränssnitt i ett systemkomponentperspektiv (alltså mellan bibliotek) och inte på den nivå vi nu ska granska. I ett mjukvarusystemperspektiv består systemets komponenter av klient, server, databas, GCM och den enkla mjukvara som hos en beacon med jämna mellanrum skickar ut information om en beacon i fråga.

I komponenterna utgör de lila rektanglarna den huvudsakliga gränsen för vår egenskrivna mjukvarulösning och visar hur denna kommer att passas in i sin kontext. Vår egen produkt kan vidare delas upp i helt egenskrivna klasser, vilka återfinns inom de orange rektanglarna inuti de lila, och klasser som tillhandahållits av exempelvis Google eller beacontillverkaren.

Högst upp till vänster ser vi kontaktytan mot användaren; denna kontaktyta utgör alltså det sociala gränssnittet. Längs med pilar finner vi i vilket format data transporteras mellan komponenter och med hjälp av vilket protokoll. Schemat har i huvudsak utgått från presentationslagret i TCP/IP-modellen och tar inte i någon högre grad mer hårdvarunära abstraktionslager i beaktande. Ett undantag är förstås beaconenheten som verkar på ett lägre, mer hårdvarunära lager.

Pilar visar åt vilket håll data huvudsakligen flyter mellan kontaktytor. Pilar som går åt två håll innebär att relevant data flyter åt bägge håll, medan pilar som bara går åt ett håll innebär att data, förutom protokollsvar och liknande, bara flyter åt ena hållet. Observera att dataflöde inte är synonymt med vilken part som initialiserar anslutningar.

Gränssnitt finns egentligen bara på punkter i diagrammet där en dataflödespil går in i en annan komponent, vilket då alltså betyder att anslutning kan göras till denna komponent genom dess gränssnitt. Dessa är markerade som streckade linjer i rött, orange eller grönt beroende på om definitionen av gränssnittet faller inom projektets ansvarsområde eller ej.

Ansvarsfördelning vid gränssnitt

I flödesschemat ser vi en rad avgränsade delar vilka alla har identifierats som komponenter i vårt mjukvarusystem. Kommunikationen mellan dessa utgår ifrån vilken typ av anslutningar som fordras varvid gränssnitt måste finnas hos den komponent man ska ansluta till.

En del av dessa gränssnitt faller helt under vårt ansvar eftersom vi på båda sidor om gränssnittet finner komponenter som ingår i den egenskrivna delen av systemet. Detta innebär att vi då både måste definiera och kunna använda gränssnittet i fråga.

I andra fall har vi delvis ansvar för gränssnittet vilket gäller då endast komponenten på den anslutande sidan om gränsen tillhör vårt projekt. I dessa fall är komponenten på andra sidan gränsen extern programvara vilken vi måste rätta oss efter i kommunikationen. Därvid måste vi alltså kommunicera med denna enligt dennes gränssnitt varför vi är skyldiga att kunna använda detta på ett korrekt sätt.

Vissa gränssnitt faller inte alls inom ramarna för vårt ansvar varvid ingen sida om ett visst gränssnitt hör till någon av våra egna komponenter i mjukvarulösningen. I dessa fall måste vi veta att kommunikationen mellan komponenterna fungerar men vi har ingen direkt skyldighet att ansvara för gränssnittet ur ett arkitekturellt perspektiv eftersom vi inte i direkt anslutning kommer att ha med det att göra. Av denna anledning kommer dessa ej att behandlas i utförligare mån.

En kommunikationsgräns med ansvar för såväl definition som användning är markerat med streckad grön linje och en helt utan ansvar är markerad med röd streckad linje. I fall då vi har ansvar för att kunna använda ett utomstående gränssnitt hos en extern komponent är dess gränssnitt markerat som en orange streckad linje.

Allmän beskrivning av relevanta komponentgränssnitt i projektets mjukvarusystem

Vår programvara är tänkt att verka både på klientnivå och servernivå, och därför är vår situation lite speciell eftersom mjukvaran inte har ett eller några avgränsade gränssnitt mot omvärlden utan även ett internt gränssnitt mellan klient och server.

Vår servermjukvara kommunicerar i huvudsak med tre komponenter:

- Googles GCM-servrar
- Lokal databasserver
- Klientapplikationer

I samtliga fall kan endast en av parterna initialisera en kontakt varför gränssnitten närmast går att definiera som en uppsättning instruktioner för hur en anslutande komponent kan använda en annan. Hade situationen varit att två olika parter kunnat initialisera kontakt med varandra oberoende av den andra hade två gränssnitt fordrats, ett i varje ände.

Vid kontakt med GCM-servern initialiseras kontakt från serverns håll, vilket resulterar i en återkoppling på direktlänk från kontaktad klientapplikation. Klientapplikationen kan dock även ta kontakt med applikationsservern för att använda dess tjänster utan att först kontaktas av GCM. Databasservern kommer inte att initialisera egna kontakter till servern, utan endast svara på förfrågningar.

Gränssnittet mot en SQL-server sköts i allmänhet genom ett Javabibliotek av SQL-rutiner kallat JDBC (med de inkluderade paketen Java.sql och Javax.sql) och i synnerhet av en drivrutin specialiserad för kommunikation mellan JDBC och just den typ av SQL-server som används. I det här fallet använder vi den drivrutin som är rekommenderad för användning ihop med MySQL vilken tillhandahålls av utvecklarna av MySQL och heter *Connector/J*.

Databasserverns gränssnitt är definierat av MySQL och sköts på låg nivå av Connector/J, och på högre nivå genom SQL-kommandon vilka specifikt i vårt fall körs genom paketet Java.sql. Servern använder lat initialisering av anslutningen till databasservern, vilket innebär att den endast initialiseras om det behövs och i så fall i form av ett globalt objekt som alla trådar får dela på. Vidare används ett enda databaskonto genom vilket servern kör sina förfrågningar.

På samma sätt som databasserverns gränssnitt bestämts av MySQL bestämmer Google gränssnittet till sina GCM-servrar. Generellt kan man ansluta via två olika protokoll, nämligen HTTP och XMPP, och vi har valt att arbeta med det förstnämnda. Google tillhandahåller

dessutom ett Javabibliotek för anslutning till sina servrar, vilket här utgör gränssnitt mellan egenskrivna serverklasser och externa serverklasser. Genom biblioteket förbereder man den JSON-sträng som därefter, genom samma bibliotek, skickas till GCM via en HTTP-anslutning.

Eftersom kontakten med GCM utmärks av att data packas som JSON-strängar, valde vi att göra på detta sätt även i vårt interna gränssnitt. Detta gör att vi kan återanvända klasser som krävs för hantering av JSON, och bidrar till en projektstandard som gör det lättare att förstå och underhålla projektet. Således förmedlas information mellan server och klient genom att först göra om informationen till en JSON-sträng vilken sedan förmedlas via en TCP-anslutning som administreras genom Javapaketet Java.net.

Kontakten mellan server och klient initialiseras av klient varvid denne förväntar sig att servern svarar enligt sin specifikation genom att skicka relevant data tillbaka. Klienten kommunicerar via en JSON-sträng som innehåller en nyckel som tillåts ha en mängd olika värden och på så sätt förmedlar vilken tjänst som klienten begär. Denna nyckel kan sägas utgöra protokollet i kommunikationen mellan klient och applikationsserver. Protokollet beskrivs detaljerat i Kapitel 4.2.4.

Komponentgränssnitt utan ansvar

I flödesschemat förekommer kontaktytor som inte ansluter direkt till någon komponent som ingår i projektets mjukvara. Dessa medför inget ansvar för vår del men gränssnitt förekommer även där, och även om vi inte ska gå in i detalj på dessa kan några saker nämnas lite kort.

GCM-servrarna kommunicerar inte någon relevant data till applikationsservern, utan använder istället data i anropet från denna för att avgöra till vilken enhet den ska skicka vidare informationen. GCM-servern ansluter alltså till gränssnittet som notifikationstjänsten på Android-enheter tillhandahåller. Denna omfattar alla applikationer som finns installerade på telefonen, och datat skickas även här som en JSON-sträng. Notifikationstjänsten undersöker vilken applikation som ska aktiveras och aktiverar sedan denna genom Android SDK, som är integrerad i klientklassen.

Ett annat viktigt gränssnitt är det som kopplar samman Bluetooth-signaler från beacons och Android-enheten. Varje beacon sänder med jämna mellanrum ut ett enkelt datapaket kallat *Bluetooth announcement*, vilket innehåller data som mottages av Bluetooth-enheten (som stödjer BLE) i telefonen. Via beacontillverkarens SDK som sömlöst integreras i våra

klientklasser kan sedan denna information föras vidare in i klientapplikationen. Det konkreta gränssnittet mellan beacon och Bluetoothenhet faller dock utanför projektets ansvarsområde.

Arkitekturella beslut

Vid de flesta gränssnitt i mjukvarusystemet faller endast kunskap om användande inom projektets ramar. Vid dessa gränssnitt har inga beslut rörande arkitektur kunnat tas eftersom komponenterna som dessa angränsar till helt och hållet avgör gränssnittets utförande varvid man som användare enbart har att anpassa sig. I det fall där ett internt gränssnitt upprättats har projektet ansvarat för gränssnittets definition varvid vi givetvis kunnat påverka dess utformning. Det som här har präglat våra beslut har bland annat varit kraven i URD (krav 3.2.4–3.2.6) om att mjukvaran ska vara snabb och inte sluka för mycket batterikapacitet.

Genom att skicka så lite data som möjligt har vi bra förutsättningar för att nå dessa krav. Datautbytet mellan komponenterna består endast av JSON-strängar som är kompakta och minneseffektiva i jämförelse med andra liknande strukturer som exempelvis XML. Listor skickas som JSON-strängar vilket innebär en väldigt liten ökning i storlek i jämförelse med hur mycket plats den faktiska informationen i en aktuell datastruktur tar. En annan anledning till att vi valt JSON-strängar är att gränssnittet mot GCM kräver JSON-strängar vilket nödvändiggjorde visst stöd för JSON i vår mjukvara. Genom att använda detta format på flera ställen kan vi återanvända klasser som vi ändå är tvungna att ha, vilket standardiserar och förenklar förståelsen för programmets struktur.

Specifika beskrivningar av externa gränssnitt med ansvar i projektets mjukvarusystem

3.1 Gränssnitt: Databas

Initialiseringskomponent: Applikationsserver
Ansvar: Användning
Dataflöde: Båda hållen
Dataformat: SQL
Kommunikationsprotokoll: Definierat av Connector/J

Servern initialiserar förfrågan till databasen genom användning av paketet Java.sql i JDBC. Detta paket har SQL-relaterade klasser och klasser som underlättar kommunikation med en SQL-server. Paketet ansluter via Connector/J till MySQL-servern och kommunikationsprotokollen består i användning av JDBC genom denna drivrutin.

3.2 Gränssnitt: Applikationsserver

Initialiseringskomponent: Applikationsklient
Ansvar: Definition och användning
Dataflöde: Båda hållen
Dataformat: JSON
Kommunikationsprotokoll: Eget protokoll förmedlat i en nyckel kallad action i JSON-strängen

Klienten initialiserar alltid kontakten med servern genom att skicka en JSON-sträng över en TCP-anslutning. JSON-strängen innehåller bland annat en nyckel kallad action vars värde anger vilken tjänst klienten vill använda sig av på servern. Denna kan sättas till en rad giltiga värden och ställer sedan i sin tur krav på övriga JSON-parametrar. Om anslutningen inte görs på detta vis, eller om datapaketet som anländer inte är en JSON-sträng med nödvändiga fält specificerade, är inte anslutningen giltig och nekas således.

Protokollet är alltså eget och tämligen enkelt; anslutningen måste ske med en JSON-sträng som måste ha en action-nyckel vilken endast får anta ett av en begränsad mängd värden. Klienten väntar sig sedan viss typ av information som svar från servern och avslutar anslutningen om svaret inte mottagits inom rimlig tid.

3.3 Gränssnitt: GCM-server

Initialiseringskomponent: Applikationsserver
Ansvar: Användning
Dataflöde: Applikationsserver till GCM-server
Dataformat: JSON
Kommunikationsprotokoll: HTTP

GCM har helt och hållet bestämt gränssnittet, vilket innebär anslutning via antingen HTTP eller XMPP. Som redan nämnts i detta kapitel har vi valt att arbeta med HTTP-lösningen. Google tillhandahåller ett klassbibliotek med vilket man kan administrera både JSON-strängen och HTTP-anslutningen som GCM förväntar sig. GCM håller reda på vilka enheter som är tillgängliga och vilka val de gjort för sina applikationer i notifikationstjänsten som finns i alla Android-enheter. Beroende på tillstånd hos mottagande enhet och inställningar skickas en notifikation till mottagaren som tas emot av enhetens notifikationstjänst, och tjänsten använder då parametrar i JSON-strängen för att avgöra hur den ska följa upp anslutningen från GCM.

4 System Design

Detta kapitel handlar om systemarkitekturen; varför vi har valt att göra som vi gjort, hur systemet fungerar, vilka begränsningar som finns, samt i detalj vilka komponenter som ingår och vad de har för uppgift. Syftet med detta kapitel är att ge läsaren en övergripande förståelse för hur systemet är tänkt att fungera, lägga fram resonemang kring valet av arkitektur, förklara vilka verktyg som har använts för att konstruera modeller och dokumentation, och slutligen ge läsaren mer detaljerad förståelse för enskilda komponenter i systemet samt hur dessa interagerar med varandra.

4.1 Design method

4.1.1 Övergripande design och arkitektur

För att komma fram till vår arkitekturella design använde vi oss initialt av brainstorming. Systemgruppen, som är ansvarig för arkitekturen, satt tillsammans och diskuterade hur arkitekturen skulle se ut. Vi antog tidigt en objektorienterad inriktning och delade upp mjukvarulösningen i två delar, server och klient, och diskuterade dessa separat. Detta föll sig naturligt eftersom de två intuitivt förefaller skilja sig åt på många sätt samtidigt som de, var och en för sig, har mycket inbördes gemensamt vad gäller funktionalitet och behov.

I det fortsatta arbetet med arkitekturen anammade vi principer från top-down-design, och vi började därför med att rita ut stora övergripande komponenter. Iterativt gick vi sedan åter till komponenterna varvid vi förfinade funktionaliteten och behoven tills vi kommit ned på den nivå då det började bli rimligt att definiera Javaklasser. Efter detta började vi identifiera och isolera olika delar av lösningen och göra en disposition av funktioner i klasser vartefter dessa ritades upp som klassdiagram på en whiteboardtavla. Databas och allt i anslutning till denna hanterades separat. I designen av databasen var det naturligt att ha två separata delar för användare respektive beacons vilka initialt utgjorde vår utgångspunkt i frågan. Efterföljande iterationer och analys av typiska SQL-frågor ledde oss vidare i arbetet mot hur dessa initiala indelningar kunde delas upp ytterligare och förfinas.

Eftersom vi kodar i Java, som är ett objektorienterat språk, var det naturligt att använda sig av en väldigt påtagligt objektorienterad design. Till att börja med ses den ytterligare abstraktion från kod som objektorienterad programmering innebär som industristandard och god programmeringssed. Dessutom, eftersom detta är ett stort projekt med många medlemmar, gör

en objektorienterad design att det blir lättare att dela upp arbetet i olika delar för att undvika att för många arbetar med samma delar. Varje användare är ansvarig för åtminstone en klass, och eftersom klasserna kan kodas oberoende av varandra (så länge man följer standarden i UML-diagrammen) kan flera klasser utvecklas parallellt vilket förkortar utvecklingstiden. Den största flaskhalsen vid programmering är annars att man måste vänta på att andra skall göra klart sitt jobb. Objektorienterad programmering är för övrigt den typ av programmering som medlemmarna i gruppen har mest erfarenhet av. De klasser som vi hade fått fram blev sedan en naturlig utgångspunkt för vilka olika komponenter som skulle beskrivas i detta dokument.

Det finns typiskt sett många fördelar med en objektorienterad design. Stora projekt har exempelvis mycket att vinna på att vara tydligt uppdelade där var och en har sitt eget ansvar och där man i enkla dokument lätt kan se vilken typ av information man kan förvänta sig från sina kollegors kod. Vidare är inkapsling, som objektorientering medför, bra eftersom det är lätt att avgöra var kod som anknyter till någon viss process finns, och modulariseringen gör såväl arbetet som koden lättare att underhålla och mer flexibelt för ändringar. Vårt intryck är att vinsten, i relation till andra designprocesser såsom exempelvis vissa agila metoder, blir större ju större projektet är.

Dock finns det negativa sidor med en objektorienterad design också. Till att börja med krävs av projektledningen och chefsprogrammeraren mycket planeringsarbete och en väldigt god översiktsförmåga så att funktionalitet placeras korrekt; ändringar i sena skeden kan ge ödesdigra konsekvenser i form av tid och kostnader. Vidare krävs en tydlig vision av det som man ska konstruera, och man bör med fördel närma sig designprocessen med ett top-down-perspektiv så att man tidigt får en helhetsbild vilket förenklar uppdelningen av funktionalitet i klasser.

Om man ska kontrastera det objektorienterade förhållningssättet med ett mer dynamiskt och agilt sådant där man låter slutprodukten växa fram under arbetets gång kan man konstatera att det objektorienterade är relativt stelt. Returtyper bör exempelvis bestämmas tidigt och kräver ändringar på många håll om dessa behöver ändras i ett senare skede. Det är också svårt att definiera och komma överens om vilken klass som ska ansvara för vad när ansvarsområden mellan klasser överlappar, vilket kan ge upphov till missförstånd mellan programmerare. Ett exempel är ListenerThread och Procedure-klasserna. Vad ska ListenerThread göra innan startProcedure anropas? Om man vill ändra på detta i efterhand kan det bli svårt att flytta funktionalitet mellan olika klasser eftersom detta även kan påverka andra klasser.

Objektorienterad design kan också försvåra modelleringen av ett problem, även om en objektorienterad design underlättar implementationen av stora projekt. Många av dessa problem kan dock lösas helt eller delvis med hjälp av ett omfattande planeringsdokument, som denna ADD, i kombination med att man tidigt vet vad kunden har för krav på produkten. Med dessa förutsättningar på plats anser vi att fördelarna överväger nackdelarna med vår valda objektorienterade designmetod.

Det kan tilläggas kan man under ett projekt som detta givetvis på mikronivå ibland avviker från den storskaliga designmetoden. Exempelvis har vi använt oss av ett agilt arbetssätt i det att vi har genererat vår UML-modell utifrån redan existerande Javadoc. Dock skrevs Javadokumentationen utifrån den modell som vi hade kommit överens om genom vår objektorienterade design.

Efter att den övergripande designen var gjord påbörjades arbetet med att skriva kodskelett och kommentarer som vi sedan synkroniserade med varandra via versionshanteringssystemet Git. När vi var klara genererade vi UML-diagram med ObjectAid, och Javadoc med Eclipse. Uppdelade UML-diagram för klient och server återfinnes i Kapitel 4.2.

4.1.2 Dokumentation

För att en objektorienterad design skall bli så effektiv som möjligt krävs genomtänkt ansvarsfördelning och tydliga gränssnitt som andra programmerare kan utnyttja i sin egen kod. Förutom detta hör som vanligt dokumentation till god programmeringssed och vi kom överens om att skriva rikligt med dokumentation för de metoder som skall användas av andra i gruppen. I URD har vi nämnt att dokumentationen skall skrivas på engelska, och detta krav gäller fortfarande, men det är accepterat att skriva noteringar till andra i gruppen på svenska om dessa kommentarer inte är permanenta. En temporär notering på svenska kan till exempel vara en fråga om huruvida en klass eller metod bör vara kvar, begäran om komplettering av dokumentation, TODO-markeringar och dylikt.

För att undvika missförstånd kring hur klasser och metoder skall användas är det viktigt att skriva korrekt och tydlig dokumentation, gärna innan metoden implementeras. Saker som bör tas upp i dokumentationen för en metod är:

1. En mening som kortfattat förklarar vad metoden gör. Detta ska stå först i dokumentationen.
2. Detaljerad beskrivning av metodens funktionalitet. Här bör man exempelvis nämna:

    a. Vad metoden normalt returnerar.
    b. Vad som kan gå fel, och hur dessa fel bör hanteras av programmeraren. Om någonting är komplicerat eller svårt att förklara i naturligt språk bör kommentaren innehålla ett kodexempel.
    c. Om metoden returnerar null, och i sådana fall när detta sker.
3. Metodanrop eller instansieringar som måste göras innan metoden anropas.
4. Hänvisningar till eventuella överlagrade metoder.
5. Om metoden använder variabler, externa resurser eller inre klasser och dessa kan ändras av användaren, bör man exempelvis nämna:
    a. Vilka variabler, externa resurser eller inre klasser som används och hur dessa påverkar ett anrop till metoden.
    b. Exempel på hur dessa variabler, externa resurser eller inre klasser genom metodanrop kan ändras av programmeraren, alternativt hänvisa till dokumentationen för dessa metoder.
6. Om metoden inte returnerar void: En tagg @return som i korta ordalag förklarar vad metoden returnerar.
7. För varje inparameter: En tagg @param som i korta ordalag förklarar vad parametern representerar.
8. För varje exception som metoden kastar: En tagg @throw som förklarar när och varför en exception kastas.

I dokumentationen för en klass bör följande tas upp:

1. En mening som kortfattat förklarar vad klassen gör. Detta ska stå först i dokumentationen.
2. En någorlunda detaljerad beskrivning av klassens funktionalitet. Det viktiga är att få en överblick; detaljerna finns ändå i dokumentationen för varje enskild metod. Några saker som är bra att nämna är:
    a. Om klassen exempelvis använder method chaining, statiska konstruktorer med speciella namn eller inte kan instansieras, och i sistnämnda fall förklara hur programmeraren kommer åt klassen.
    b. Övergripande egenskaper, exempelvis om klassen implementerar ett visst gränssnitt, ärver från andra klasser, ska startas i en ny tråd eller startar egna trådar.

    c. Om andra klasser måste instansieras eller om vissa statiska metoder måste köras innan klassen skapas eller används, specificera hur detta kan göras, gärna med kodexempel.
    d. Ge gärna ett kodexempel på hur klassen är tänkt att användas, speciellt om felhantering är krångligt eller ett typiskt användande av klassen involverar komplicerade metodanrop eller kritiska logiska flöden där metoder måste anropas i en viss ordning.
3. En tagg @version som innehåller versionnumret för den senaste versionen av klassen.
4. För varje programmerare som har gjort betydande insatser: En tagg @author som innehåller namnet på personen.
5. Om en klass har gjorts av flera programmerare, lägg gärna till versionhistorik. Detta är inte kritiskt eftersom vi använder Git, men kan underlätta för att minska risken för missförstånd.

4.1.3 Småskaliga designmönster

På en lägre abstraktionsnivå i vår design har vi använt oss av följande principer:

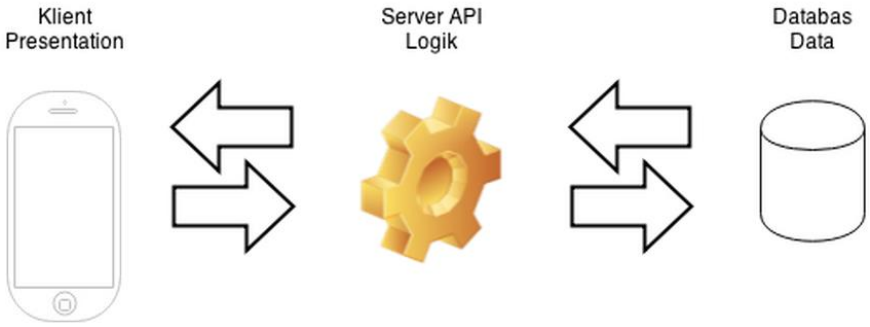
1. Klient-server-modellen
2. Three-tier architecture
3. Singleton-klasser
4. Fluent interface
5. Façade

4.1.3.1 Klient-server-modellen

Förutom att innebära ett uppenbart objektorienterat ställningstagande innebär faktumet att man delar upp en mjukvarulösning i de tydliga komponenterna klient och server att den så kallade klient-server-modellen bör implementeras. Denna syftar till att ett tydligt gränssnitt för kommunikation och informationsutbyte mellan dessa ska finnas. Klienten, å ena sidan, hanterar avsökning av beacons och presenterar användaren med ett intuitivt användargränssnitt som gör att användaren kan interagera med servern. Servern, å andra sidan, håller reda på inloggningar, bokföring, användardetaljer, inställningar och andra data som skall delas mellan olika enheter. Gränssnittet mellan dessa är tydligt formulerat och bygger på JSON-strängar med vissa nycklar som mer konkret utgör protokollet mellan de två.

4.1.3.2 Three-tier architecture

Vi kommer att använda oss av *three-tier architecture* för systemet. En three-tier architecture delas upp i tre lager: Presentation, logik och data. Figuren nedan illustrerar situationen. Data består av en databas, logik består av vårt server-API och presentation består av klientens användargränssnitt. Servermjukvara och databas kommer att köras på en och samma serverhårdvara medan klienten körs på användarnas handhållna Android-enheter.



4.1.3.3 Singleton-klasser

Klasserna DBConn och ServerCon lät vi vara singleton. En klass som är singleton är statisk och kan således bara instansieras en gång. På så sätt undviker vi förvirring kring vilket objekt som ska användas, och eftersom objektet är statiskt är det tillgängligt för alla klasser utan att man behöver en referens till objektet. Detta är speciellt smidigt i Android där ett objekt som ServerCon annars skulle behöva skickas mellan olika Activities. Om ServerCon inte skulle vara statisk, skulle man behöva göra klassen till exempelvis en Bundle, vilket är tämligen komplicerat. Syftet med ServerCon är att den skall vara tillgänglig för alla klasser, och även om statiska klasser och metoder generellt sett anses vara dåligt kan det finnas situationer där fördelarna är större än nackdelarna. I just detta fall kändes det naturligt att göra ServerCon statisk, eftersom det blir lättare för andra programmerare i gruppen att använda den.

Till skillnad från en statisk klass, instansieras en singleton-klass först när man använder den för första gången. Instansieringen sker inte genom att kalla på dess konstruktor (som är privat) utan genom att anropa exempelvis metoden getInstance som returnerar klassens aktuella instans om en sådan finns, eller skapar en ny om det är första gången som metoden getInstance anropas. Detta gör att en singleton-klass är mer minnesbesparande än motsvarande statiska klass, så länge den inte används.

4.1.3.4 Fluent interface

Ett annat designmönster som vi använder är fluent interface som implementeras med method chaining. [1] Method chaining är ett smidigt sätt att kombinera setters i Java för att sätta parameterar. Syftet med ett fluent interface är att skriva ett API på sådant sätt att konstruktion av objekt liknar naturligt språk. Vi illustrerar detta med ett litet kodexempel. I stället för att skapa ett objekt och sätta varje parameter individuellt på följande sätt ...

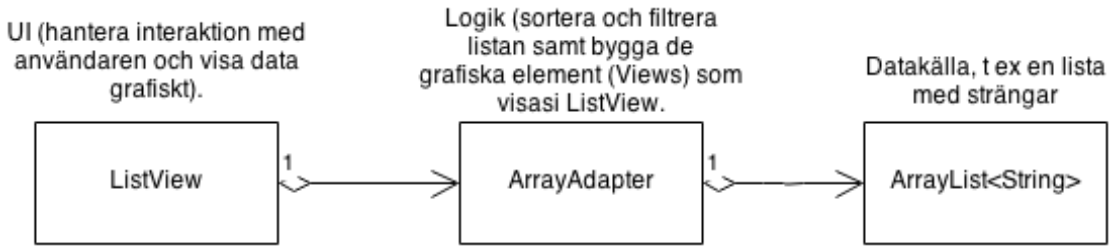
```
Beacon beacon = new Beacon();
beacon.setUUID(“E13A-4334-6H42-4HTE2”);
beacon.setBatteryPercent(95);
beacon.setMajor(24224);
beacon.setMinor(32323);
```

… så kan man kedja ihop funktionerna med ett metodanrop på följande sätt ...

```
new Beacon.BeaconFactory().setUUID(“E13A-4334-6H42-4HTE2”).
            setBatteryPercent(95).
            setMajor(24224).
            setMinor(32323).
            build();
```

Detta skrivsätt är vanligt bland annat i bibliotek från Google. Först anropas en statisk inre klass som är ansvarig för att bygga objektet. Varje setter kedjas ihop med hjälp av method chaining. På slutet anropas typiskt sett en metod build som returnerar själva objektet. Därför kallas method chaining ibland för The Builder Pattern.

Android använder sig även av MVC (Model-View-Controller) som vi kommer att behöva implementera i form av en ListView respektive ArrayAdapter. Detta illustreras i figuren nedan. För att skapa en scrollista i Android behöver man skapa en ListView (View), som är ett UI-element i Android som användaren kan interagera med. Innehållet i en ListView hämtas från en ArrayAdapter (Controller) som hanterar sökning, sortering och filtrering av listan samt hur data presenteras på skärmen. En ListAdapter hämtar i sin tur data från en datastruktur (Model), exempelvis en ArrayList med strängar. En ListView respektive ListAdapter kommer användas i klasserna SearchFragment, FavoritesFragment och LogbookFragment.



4.1.3.5 Façade

Namnet på detta designmönster kommer från det arkitektoniska begreppet fasad som syftar på framsidan på en byggnad. Façade innebär att man förser ett API till ett programpaket eller bibliotek med ett enklare gränssnitt som bevarar potentialen hos den underliggande strukturen men gör den mer lättillgänglig och anpassar den för aktuellt ändamål. Rent konkret skapas alltså en klass som utgör ett yttre API (“fasad”) som kommunicerar med det inre, men som genom att innehålla skräddarsydda metoder med intuitiva namn underlättar för andra programmerare att på ett snabbt och effektivt sätt få åtkomst till komplicerade bibliotek som används i stor utsträckning.

Exempel på denna typ av klass i vårt projekt är klasserna DBConn och ServerCon vilka båda sköter komplicerade uppgifter i stora bibliotek och tillhandahåller dessa funktioner som en del av ett enklare API som projektgruppen använder sig av. DBConn gör allt från att förbereda databasförfrågningar till att sköta själva anslutningen till denna via paketet Java.sql, medan ServerCon tillåter smidig interaktion med paketet Java.net som i sig innehåller mer komplicerade metoder för nätverkstjänster. Förutom att göra arbetet enklare för programmeraren gynnar även detta designmönster ett objektorienterat förhållningssätt då façade ytterligare kapslar in funktionalitet på ett intuitivt sätt.

4.1.4 Namnkonventioner

För att undvika missförstånd och skapa “ren” kod är det viktigt att komma överens om en namnkonvention. I Java-koden har vi använt oss av standardkonventionen UpperCamelCase för klasser och lowerCamelCase för metoder och variabler. Klasser har namn som innehåller den klass från vilken de ärver eller hör till. Exempelvis kan en klass som ärver från Activity heta ClassNameActivity och en klass som implementerar gränssnittet Procedure heta ClassNameProcedure.

Databastabellerna skrivs i lowercase och understreck används i stället för mellanslag, som exempelvis table_with_long_name.

Setters och getters bör typiskt sätt ha namn som börjar på set och get, som exempelvis getMyVariable eller setConnectionListener.

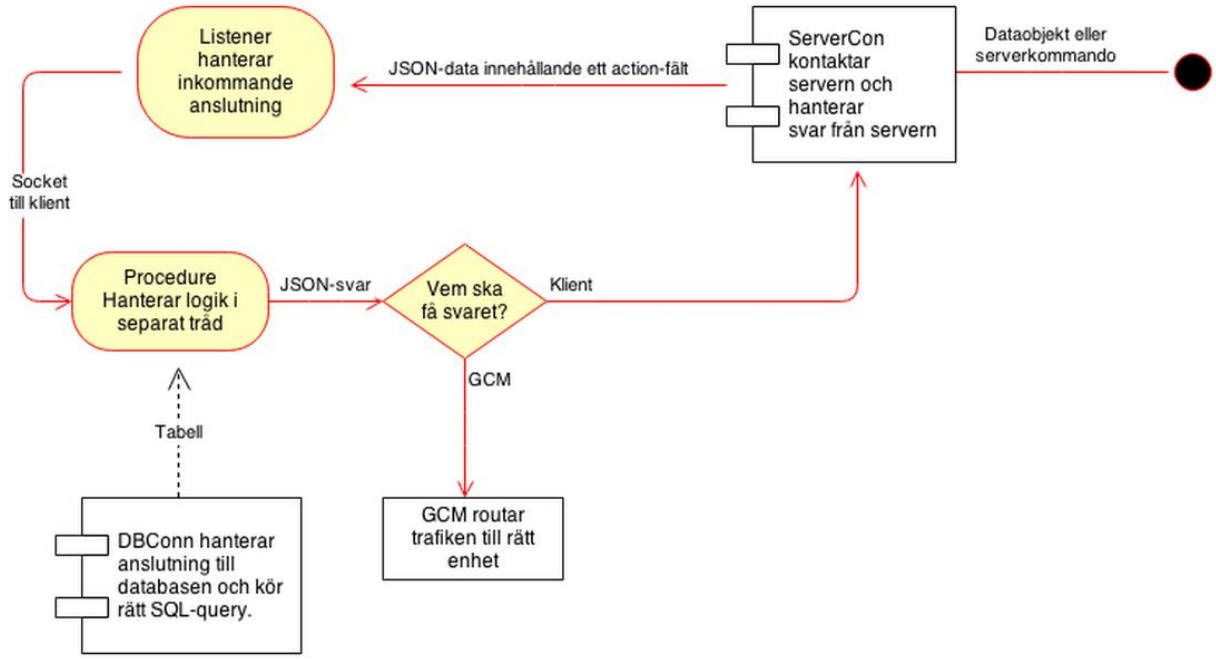
4.2 Decomposition description

4.2.1 Översikt

De viktigaste komponenterna på serversidan är:

- Klassen Listener, som innehåller en main-metod.
- Den grupp med klasser som implementerar gränssnittet Procedure.
- Klassen DBConn, som hanterar anslutningen till databasen.

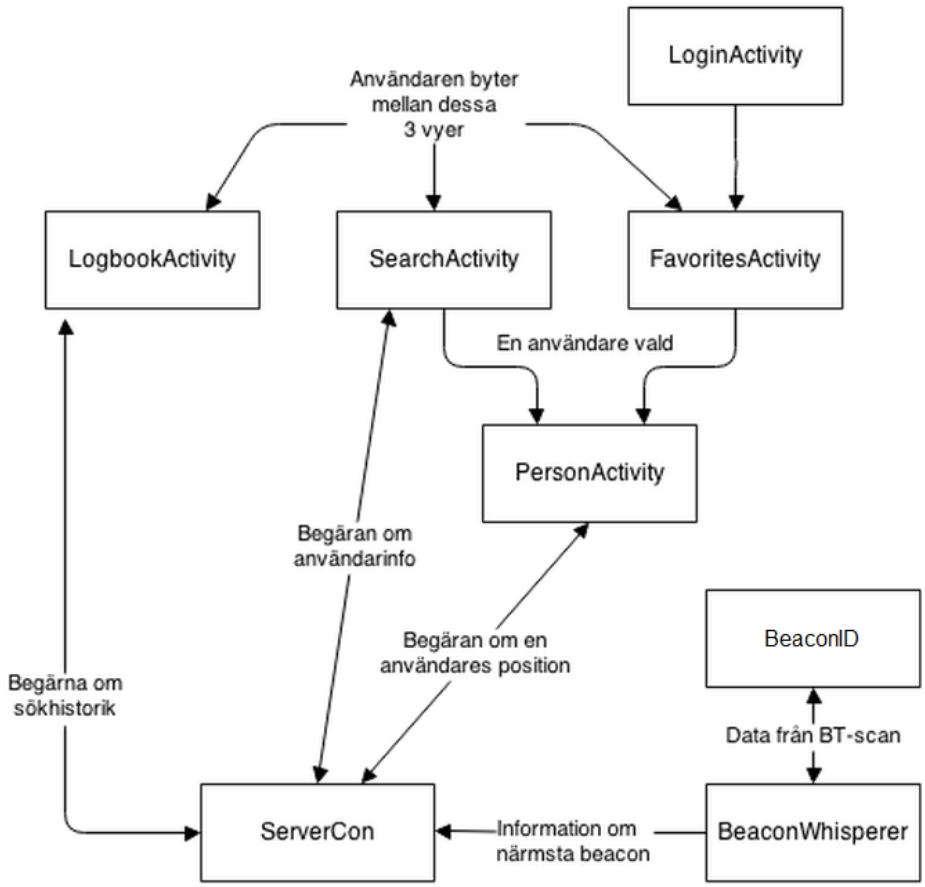
Nedan illustreras en översikt över dessa serverkomponenter i ett diagram.



De viktigaste komponenterna på klientsidan är:

- De klasser som hanterar UI:et (bland annat LoginActivity, FavoritesFragment och SearchFragment).
- Klassen ServerCon som hanterar kommunikation med servern och BeaconWhisperer.
- Klassen BeaconWhisperer som söker efter och lagrar närmaste beacon.

Nedan illustreras en översikt över dessa klientkomponenter i ett diagram.



4.2.2 Detaljerad beskrivning

Systemet vi skall konstruera bygger på en klient-server-modell som kommunicerar med varandra över TCP/IP. Klienten kommer bestå av en handhållen enhet som kör Android 4.3 eller senare. Klienten kör en applikation som samlar in information om närliggande beacons, hanterar kommunikation med servern och ger användaren tillgång till ett gränssnitt som presenterar information för användaren enligt de användarkrav som finns specificerade i Kapitel 3 i URD.

Servern består av två delar, data och logik, som tillsammans med klienten utgör den three-tier-architecture som i föregående underkapitel beskrevs. Data består av en databas som hanterar all information som systemet måste hålla koll på, som exempelvis beacons och användaruppgifter. Logiken, som är skriven i Java, är själva mjukvaran som körs på servern. Denna mjukvara utgör i grova drag ett API till klienten. Huvudsyftet med servermjukvaran är att fungera som en brygga mellan databasen och klienten. Detta implementeras med hjälp av klasserna Listener, DBConn och ServerCon.

Listener är en klass som innehåller kod för att starta serverdemonen och lyssna på inkommande anslutningar. Vid en inkommande anslutning startas en ny tråd varpå kod körs för att hantera kommunikationen med klienten, det vill säga skicka och ta emot data enligt protokollet. Tråden hanteras i den inre klassen ListenerThread som ärver från Thread. Denna klass innehåller en metod run som körs när man anropar metoden start på tråden. Varje gång servern måste plocka fram data ur databasen används klassen DBConn som hanterar förfrågningar till databasen.

Nedan visas ett kodutdrag som förklarar flödet för hur inkommande anslutningar hanteras.

```
ServerSocket serverSocket = createServerSocket();
if (serverSocket == null) {
      debug("Panic! Could not create server socket. Aborting...");
      System.exit(1);
}
while (true) {
      try {
            debug("Listening for incoming connections...");
            Socket clientSocket = serverSocket.accept();
            // Handle connection in a separate thread.
            ListenerThread thread = new ListenerThread(clientSocket);
            thread.start();
      } catch (IOException e) {
            debug("I/O exception occured in main, retrying...");
            e.printStackTrace();
      }
}
```

Metoden debug är en statisk metod i Listener som används av alla klasser på servern. Metoden gör en debugutskrift till stdout om debugflaggan är påslagen. Denna flagga kan slås på genom att anropa serverdemonen med argumentet -d, som exempelvis genom kommandot `java -jar flaskpost.jar -d`. Att ha en debugmetod är viktigt för att kunna följa det logiska flödet på servern och felsöka eventuella fel. Eftersom vi kör servermjukvaran på en extern dator har vi inte möjlighet att köra en debugger och stega oss igenom koden med breakpoints.

Syftet med ListenerThread, som är en lokal klass i Listener, är att hantera anslutningar i en separat tråd. Så fort en anslutning har tagits emot skapas en ny instans av ListenerThread som hanterar kommunikationen med klienten. På så sätt kan vi hantera flera anslutningar på samma gång samtidigt som svarstiden minskar.

Socketen som skapas skickas vidare till ListenerThread som hämtar data från klienten. Dessa data som klienten skickar är en JSON-sträng som innehåller ett fält *action* (se Kapitel 4.2.4). Detta fält specificerar vad klienten behöver för information, som exempelvis en användares position eller sin logg. Fältet action parsas och rätt procedur startas beroende på vilket värde som erhålls. En procedur är en kommunikationssekvens med klienten där server och klient skickar data enligt protokollet.

Varje procedur abstraheras med hjälp av en klass som implementerar gränssnittet Procedure. Dessa klasser heter enligt namnkonventionen LoginProcedure, SearchProcedure, FavoritesProcedure, PositionRequestProcedure, PositionResponseProcedure och LogProcedure. Alla Procedure-klasser innehåller metoden startProcedure som startar kommunikationssekvensen med klienten givet en socket. En referens till socketen i ListenerThread skickas alltså vidare till Procedure-klassen och startProcedure anropas.

När kommunikationen med klienten är klar stängs socketen och samlas upp av Javas skräpsamlare. Socketen för en klient återanvänds alltså inte för framtida användning utan slängs bort. Anledningen till detta är att det skulle bli onödigt mycket administrativt arbete att hålla kolla på alla anslutningar och vilka anslutningar som fortfarande är öppna. Problem uppstår dock när svar ska skickas till klienten efter en positionsförfrågan. Låt oss illustrera problematiken med ett exempel:

1. A begär B:s position genom att kontakta servern.
2. Servern kontaktar B med hjälp av GCM, varpå anslutningen till A stängs.
3. B svarar med sin position till servern, men servern har nu tappat bort anslutningen till A.
4. A får inget svar från servern och gör timeout.

Problemet grundar sig i problemet att inkommande anslutningar från B hanteras av klassen Listener medan svaret som ska skickas tillbaka till A hanteras av en Procedure-klass som dessutom körs i en separat tråd. För att lösa detta problem väljer vi att inte slänga bort sockets som skapas när en klient begär en annan användares position. Efter att förfrågan skickas till B via GCM sparas anslutningen till A i en HashMap som heter pendingRequests. Denna HashMap innehåller alla pågående anslutningar till klienter som väntar på svar efter att ha genomfört en positionsförfrågan. Nyckeln i denna HashMap är hashen för strängen A:B eftersom denna bör vara unik.

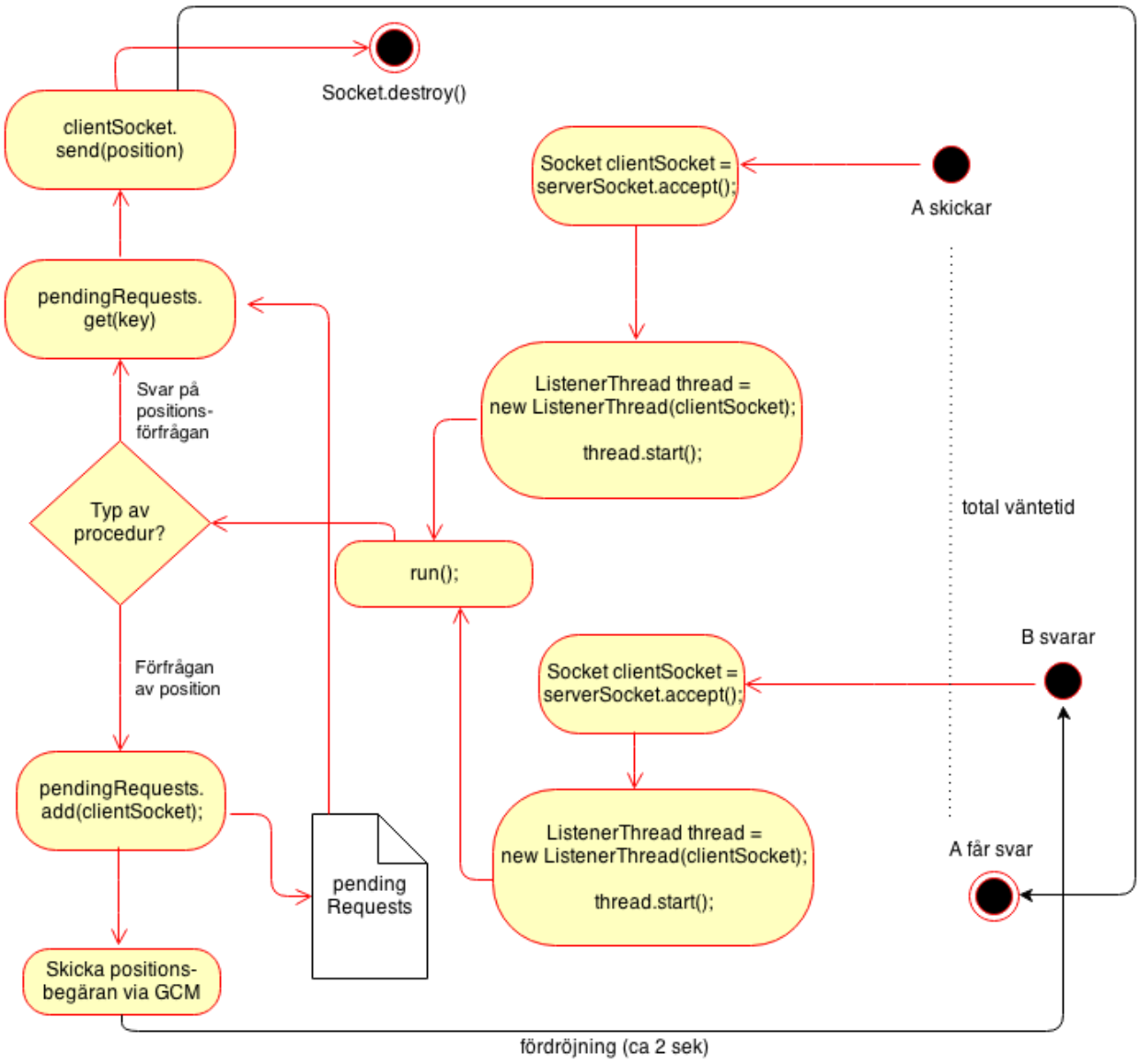
När B svarar med sin position hanteras denna förfrågan som tidigare i en separat tråd. Rätt anslutning plockas fram ur pendingRequests om socketen fortfarande är öppen, det vill säga om klienten inte har gjort timeout. Svaret skickas på socketen varpå den kastas bort, och kommunikationen med klienten är då klar. Notera alltså att klienten måste implementera en timeout. Från dokumentationen i ServerCon kan vi läsa:

> *The server will not answer if the user is offline, so you should specify a timeout. If you don't want to specify a timeout manually, you can use the overloaded method of requestPosition without timeout parameter.*

Om B är offline, exempelvis på grund av att användarens internetuppkoppling är ur funktion eller på grund av ett internt fel i telefonen kommer servern, och därmed A, inte få något svar. Vilken timeout som bör användas har inte bestämts än. Just nu är parametern default_timeout satt till fem sekunder, men det kan komma att ändras beroende på hur snabbt det går för telefonen att svara.

Den observanta läsaren har säkert noterat att det finns en potentiell minnesläcka här. Om klienten inte svarar kommer en socket att sparas i pendingRequests, men den tas aldrig bort. Det blir antagligen svårt för en enskild användare att utnyttja detta för att göra krascha servern eftersom varje användare bara kan skapa lika många sockets i pendingRequests som det finns användare online, men det är ändå en brist i vår design. Man kan lösa detta genom att implementera en egen skräpsamlare som går igenom pendingRequests med jämna intervall och tar bort sockets där anslutningen har stängts av klienten. En mer avancerad variant är att ge varje socket en tidsstämpel och ta bort sockets som är äldre än ett visst antal sekunder.

Nedan illustreras ett flödesschema för positionsförfrågan och svar med hjälp av pendingRequests.

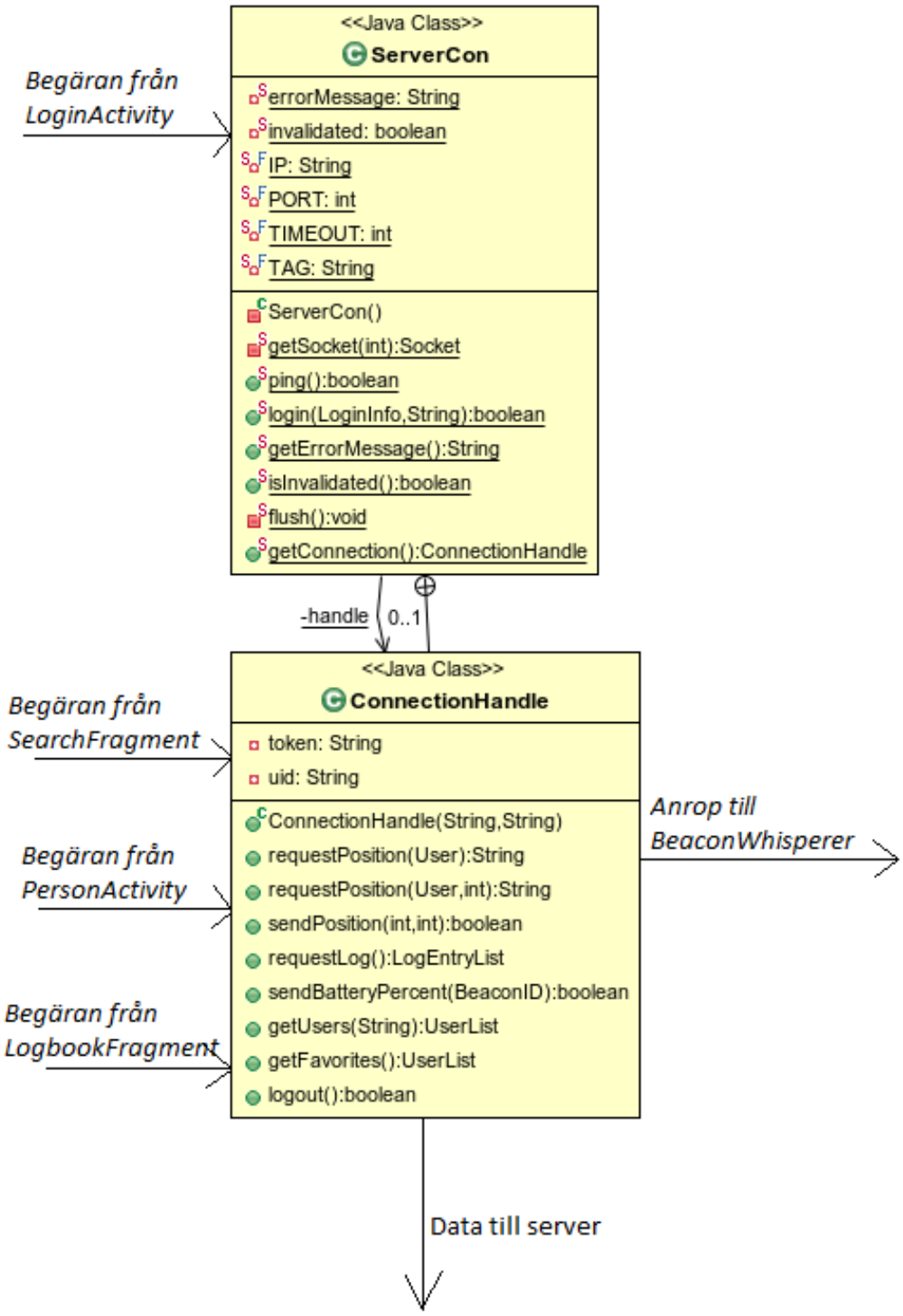


DBConn använder MySQL JDBC för att prata med databasen och köra rätt SQL-fråga. Fördelen med DBConn är att det blir enkelt att plocka ut information ur databasen eftersom man endast behöver anropa en metod, som exempelvis getUsers för att få alla användare ur databasen, i stället för att skapa en SQL-fråga och köra denna direkt mot databasen. På så sätt så slipper vi skriva samma kodrader på flera ställen (kodduplicering) och den som skriver koden behöver inte hålla koll på tabellerna i databasen för att SQL-frågan skall bli korrekt, som exempelvis vilka tabeller som innehåller vad eller vilka joins som behöver göras. Dessutom slipper vi riskerna med att koden slutar fungera om databasdesignen görs om eftersom SQL-frågorna i sådana fall måste ändras.

DBConn innehåller också metoder för att konvertera från klassen ResultSet som innehåller en tabell från en databas, till en lista med nyckel-värde-par av typen HashMap. Denna lista används av ServerOutputFormatter som skapar en JSON-sträng vilken sedan kan skickas på socketen. Låt oss ta ett exempel: Om klienten begär en lista med inloggade användare görs först en förfrågan från klient till server. Servern plockar ut användare från databasen med hjälp av DBConn. DBConn plockar ut ett ResultSet som konverteras till List<HashMap<String, String>>. Detta objekt skickas till ServerOutputFormatter som omvandlar listan till en JSON-sträng, vilken sedan skickas på socketen. Klienten på andra sidan läser JSON-strängen, och ett nytt UserList-objekt skapas genom att instansiera ett nytt UserList-objekt med JSON-strängen som parameter till konstruktorn.

ServerCon är en klass på klientsidan som hanterar kontakten med servern, bland annat genom att skicka och ta emot användarpositioner samt begära inloggade användare och loggbok. Fördelen med detta förfarande är att ServerCon hanterar saker som klientsidans utvecklare inte ska behöva ha kunskap om, som exempelvis protokoll, certifikat och att hålla koll på kontaktuppgifter för servern. Utan klassen ServerCon står man inför samma problem som på serversidan, nämligen kodduplicering och skör kod som riskerar att krascha när exempelvis IP-adressen, protokollet eller certifikatet till servern ändras.

Nedan illustreras ServerCon som en komponent med dess lokala klass ConnectionHandle som innehåller API:et.


<<Java Class>>
ServerCon
errorMessage: String
invalidated: boolean
IP: String
PORT: int
TIMEOUT: int
TAG: String
ServerCon()
getSocket(int):Socket
ping():boolean
login(LoginInfo,String):boolean
getErrorMessage():String
isInvalidated():boolean
flush():void
getConnection():ConnectionHandle
Begäran från LoginActivity
-handle
0..1
<<Java Class>>
ConnectionHandle
token: String
uid: String
ConnectionHandle(String,String)
requestPosition(User):String
requestPosition(User,int):String
sendPosition(int,int):boolean
requestLog():LogEntryList
sendBatteryPercent(BeaconID):boolean
getUsers(String):UserList
getFavorites():UserList
logout():boolean
Begäran från SearchFragment
Begäran från PersonActivity
Begäran från LogbookFragment
Anrop till BeaconWhisperer
Data till server

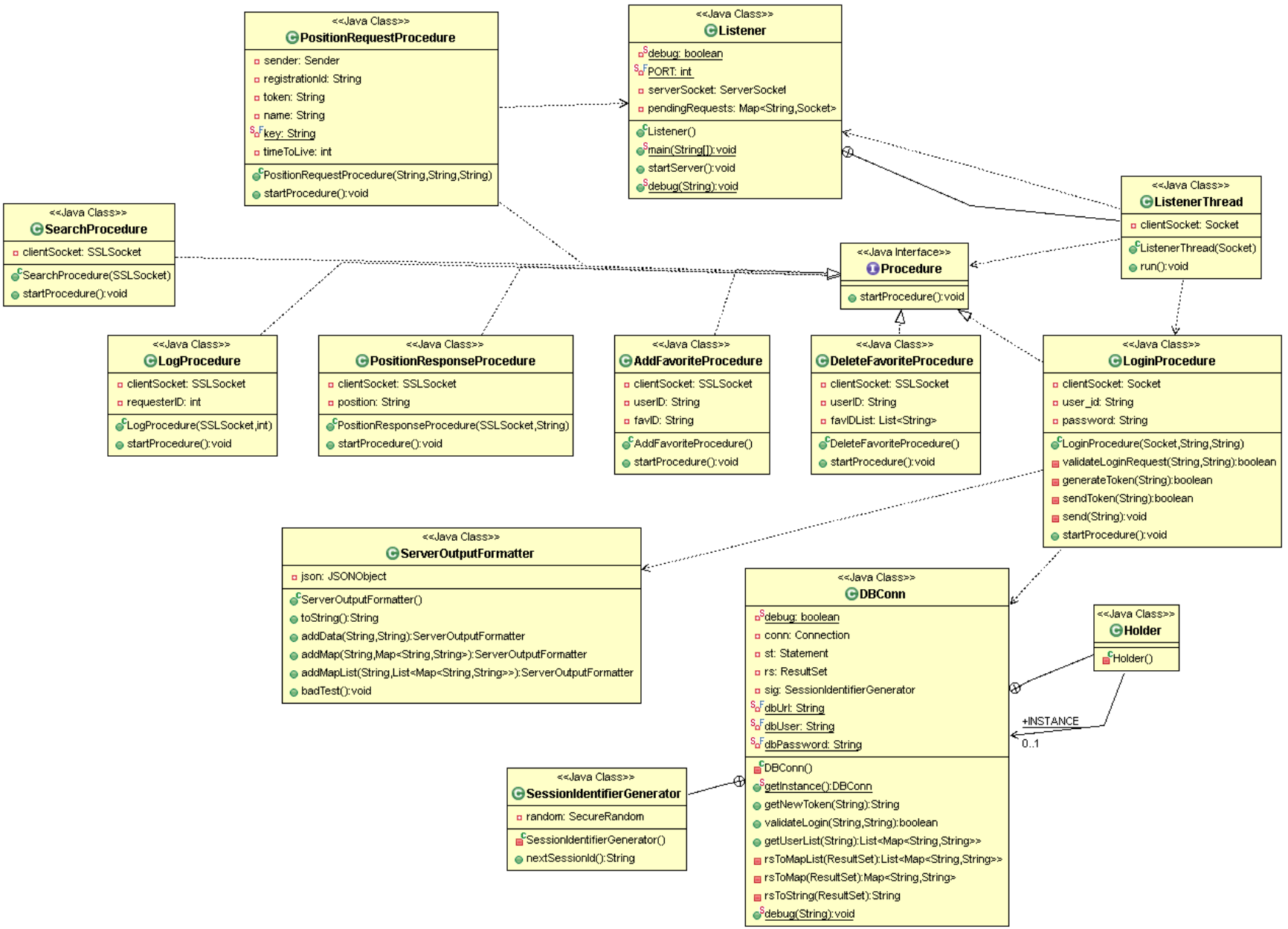
Procedure-klasserna följer protokollstandarden som detaljerat specificeras i Kapitel 4.2.4.

Fördelarna med en klient-server-modell är att det underlättar kommunikationen mellan användare, men också att det blir enklare att göra systemet säkert. Utan en central enhet som håller koll på vilka användare som får använda systemet blir det omöjligt att sätta upp någon typ av autentisering. Detta skulle medföra att vem som helst kommer kunna använda systemet, vilket inte är förenligt med våra användarkrav. Dessutom blir det klumpigt att hantera förfrågningar av position utan en server som dirigerar trafiken eftersom alla klienter måste hålla koll på varandra. Ännu en nackdel skulle vara att klienten måste ha information om alla beacons vilket skulle ta plats på telefonen. Så fort nya beacons läggs till i systemet måste klienten uppdateras vilket blir ohållbart. Slutsatserna av denna analys är alltså att en klient-server-modell är den mest passande modellen för vårt system.

Kommunikationen mellan klient och server ska som tidigare nämnts ske med hjälp av protokollstacken TCP/IP som är standard idag. Kommunikationsprotokollet bygger på JSON och specificeras i Kapitel 4.2.4. För att avkoda JSON-strängarna använder vi ett tredjepartsbibliotek som heter json-simple.

Data som skickas sänds som enskilda strängar. Varje socket har en inström och en utström som erhålls med hjälp av metoderna getInputStream respektive getOutputStream. Dessa strömmar kan sedan användas för att skapa en BufferedReader för att läsa från socketen, och en PrintWriter för att skriva till socketen.

Nedan illustreras en detaljerad översikt av alla komponenter på serversidan.


<<Java Class>>
PositionRequestProcedure
sender: Sender
registrationId: String
token: String
name: String
key: String
timeToLive: int
PositionRequestProcedure(String,String,String)
startProcedure():void
<<Java Class>>
Listener
debug: boolean
PORT: int
serverSocket: ServerSocket
pendingRequests: Map<String,Socket>
Listener()
main(String[]):void
startServer():void
debug(String):void
<<Java Class>>
ListenerThread
clientSocket: Socket
ListenerThread(Socket)
run():void
<<Java Class>>
SearchProcedure
clientSocket: SSLSocket
SearchProcedure(SSLSocket)
startProcedure():void
<<Java Interface>>
Procedure
startProcedure():void
<<Java Class>>
LogProcedure
clientSocket: SSLSocket
requesterID: int
LogProcedure(SSLSocket,int)
startProcedure():void
<<Java Class>>
PositionResponseProcedure
clientSocket: SSLSocket
position: String
PositionResponseProcedure(SSLSocket,String)
startProcedure():void
<<Java Class>>
AddFavoriteProcedure
clientSocket: SSLSocket
userID: String
favID: String
AddFavoriteProcedure()
startProcedure():void
<<Java Class>>
DeleteFavoriteProcedure
clientSocket: SSLSocket
userID: String
favIDList: List<String>
DeleteFavoriteProcedure()
startProcedure():void
<<Java Class>>
LoginProcedure
clientSocket: Socket
user_id: String
password: String
LoginProcedure(Socket,String,String)
validateLoginRequest(String,String):boolean
generateToken(String):boolean
sendToken(String):boolean
send(String):void
startProcedure():void
<<Java Class>>
ServerOutputFormatter
json: JSONObject
ServerOutputFormatter()
toString():String
addData(String,String):ServerOutputFormatter
addMap(String,Map<String,String>):ServerOutputFormatter
addMapList(String,List<Map<String,String>>):ServerOutputFormatter
badTest():void
<<Java Class>>
DBConn
debug: boolean
conn: Connection
st: Statement
rs: ResultSet
sig: SessionIdentifierGenerator
dbUrl: String
dbUser: String
dbPassword: String
DBConn()
getInstance():DBConn
getNewToken(String):String
validateLogin(String,String):boolean
getUserList(String):List<Map<String,String>>
rsToMapList(ResultSet):List<Map<String,String>>
rsToMap(ResultSet):Map<String,String>
rsToString(ResultSet):String
debug(String):void
<<Java Class>>
Holder
Holder()
+INSTANCE
0..1
<<Java Class>>
SessionIdentifierGenerator
random: SecureRandom
SessionIdentifierGenerator()
nextSessionId():String

Arkitekturen på klientsidan utgörs av ett antal klasser som hanterar gränssnitt och logik. De viktigaste klasserna på klientsidan är LoginActivity, PersonActivity, MainActivity, LogbookFragment, SearchFragment, FavoritesFragment, BeaconWhisperer och ServerCon (vars funktion har förklarats tidigare i texten). Klasser vilkas namn slutar på Activity ärver från Activity-klassen och dessa klasser hör ihop med en XML-fil som definierar gränssnittet för en vy på telefonen. I dokumentationen av Android SDK står det om klassen Activity:

> *An activity is a single, focused thing that the user can do. Almost all activities interact with the user, so the Activity class takes care of creating a window for you in which you can place your UI with setContentView(View). While activities are often presented to the user as full-screen windows, they can also be used in other ways: as floating windows (via a theme with windowIsFloating set) or embedded inside of another activity (using ActivityGroup).* [3]

LoginActivity är den aktivitet som öppnas när applikationen startas. Som namnet antyder ansvarar LoginActivity för inloggning av användaren. LoginActivity är ansvarig för att instansiera ServerCon. Detta görs genom att anropa metoden login som skapar en ConnectionHandle som man kan använda för att skicka och ta emot data från servern. I dokumentationen för ServerCon står det:

> *ServerCon is responsible for handling all communication with the server. A connection can be created using the static method login. Once a connection is established, use getConnection() to get access to the API.*
>
> *This method is also static, which means that once you have an open connection to the server, you're all set. There is no need to pass the connection handle between activities as a bundle.*

Designen är relativt enkel och innehåller två textfält där användaren kan mata in användarnamn respektive lösenord. Det finns också möjlighet att bocka i kryssrutan “Kom ihåg mig” varpå klienten sparar inloggningsuppgifterna lokalt på Android-enheten. Detta sker genom att serialisera ett User-objekt och spara det på exempelvis enhetens lagringsminne. På så vis slipper man mata in inloggningsuppgifterna vid varje inloggning vilket sparar tid. Detta kan delvis hänföras till URD-kravet att applikationen skall vara “snabb”, men också att det inte skall finnas några distraherande moment. Många kan säkerligen uppleva det som störande att behöva ange sina inloggningsuppgifter varje gång de ska använda applikationen, och därför är detta en funktion som vi anser vara mycket viktig.

Vid inloggning skickas användaren vidare till MainActivity som fungerar som en platshållare för tre flikar - sökning, favoriter och loggbok. Varje sådan flik implementeras med hjälp av klasser som ärver från Fragment. Ett fragment kan beskrivas som en del av en aktivitet:

> *A Fragment represents a behavior or a portion of user interface in an Activity. You can combine multiple fragments in a single activity to build a multi-pane UI and reuse a fragment in multiple activities. You can think of a fragment as a modular section of an activity, which has its own lifecycle, receives its own input events, and which you can add or remove while the activity is running (sort of like a "sub activity" that you can reuse in different activities).* [4]

I vårt fall har vi tre fragments:

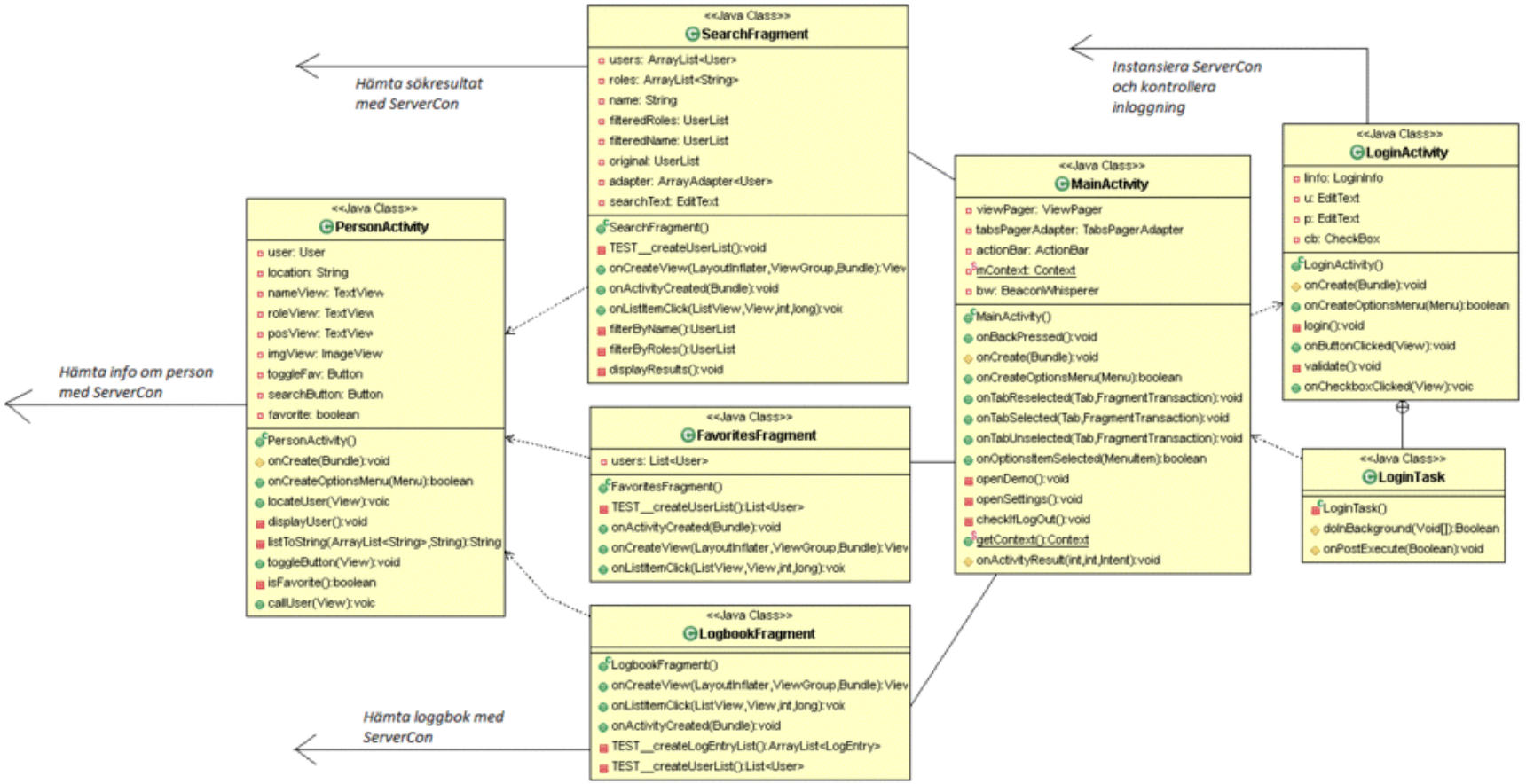
- SearchFragment
- FavoritesFragment
- LogbookFragment

SearchFragment är den flik där användaren kan välja att söka efter andra användare. Denna klass skall ladda ner en lista med användare från servern som matchar ett givet sökkriterium när en sökning påbörjas och presentera resultatet i en lista. Listan implementeras med hjälp av en ListAdapter som hämtar data från en UserList som hämtas från servern. En UserList innehåller en lista med User-klasser. User-klassen fungerar som en platshållare som innehåller data om en användare. När användaren trycker på listan öppnas en PersonActivity som visar information om vald användare, bland annat användarens position. Metoden onCreate i PersonActivity gör en positionsförfrågan med metoden requestPosition i ServerCon och visar resultatet i exempelvis en TextView.

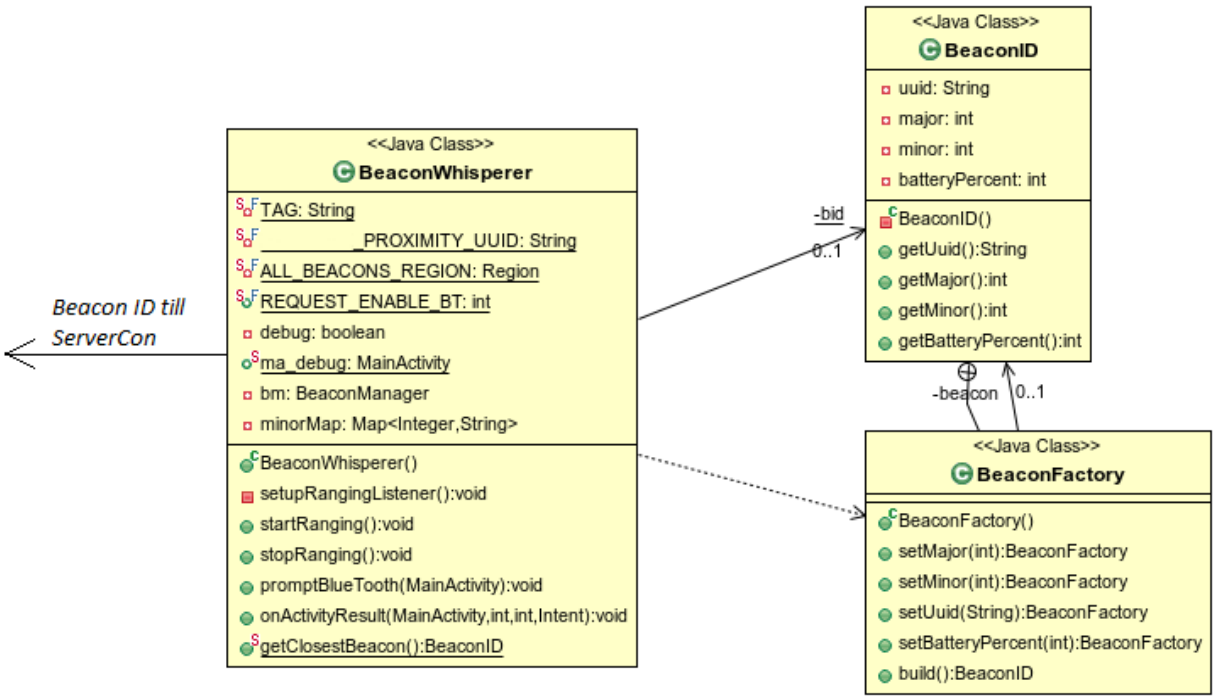
FavoritesFragment visar en lista med favoritanvändare. Vem som klassas som favoritanvändare har vi inte bestämt än; det kan exempelvis vara anställda som arbetar på samma avdelning eller anställda som användaren själv markerar som favorit. Man skulle också kunna tänka sig en kombination av dessa två. Även FavoritesFragment implementeras lämpligtvis med hjälp av en ListAdapter. Informationen i listan hämtas från servern när MainActivity startas för första gången. Man kan också tänka sig att det vore bra att implementera en uppdateringsfunktion så vyn kan laddas om när man lägger till nya favoriter och således slipper starta om applikationen. Tanken är att favoriter ska sparas i databasen eftersom man då kan byta telefon och ändå ha sina favoriter kvar. Favoriter hämtas med hjälp av metoden getFavorites i ServerCon. Denna metod returnerar, precis som getUsers i ServerCon, ett UserList-objekt.

LogbookFragment visar en logg med positionsförfrågningar som har gjorts på användaren under en i förväg bestämd tidsperiod, som exempelvis de senaste dagarna. Även denna bör implementeras med en ListAdapter. Varje post i loggen skall innehålla en tidsstämpel när förfrågan gjordes och namnet på den användare som gjorde förfrågan. Loggen hämtas med hjälp av getLog i ServerCon som returnerar en LogEntryList. LogEntryList är en klass som innehåller en ArrayList med LogEntry-klasser och en konstruktor som tar JSON-data som inparameter.

Nedan illustreras hur arkitekturen för API:t i vår applikation ser ut. Vårt API kan delas upp i tre lager. Det första lagret, längst till höger i figuren, är inloggning som implementeras genom klasserna LoginActivity och LoginTask. Andra lagret, i mitten av figuren, är MainActivity med de tre flikarna SearchFragment, FavoritesFragment och LogbookFragment som man kan bläddra mellan. Tredje lagret, längst till vänster i figuren, är PersonActivity som öppnas när användaren har tryckt på en användare i de tre flikarna.


«Java Class»
SearchFragment
users: ArrayList<User>
roles: ArrayList<String>
name: String
filteredRoles: UserList
filteredName: UserList
original: UserList
adapter: ArrayAdapter<User>
searchText: EditText
SearchFragment()
TEST__createUserList():void
onCreateView(LayoutInflater,ViewGroup,Bundle):View
onActivityCreated(Bundle):void
onListItemClick(ListView,View,int,long):voic
filterByName():UserList
filterByRoles():UserList
displayResults():void
Hämta sökresultat med ServerCon
Instansiera ServerCon och kontrollera inloggning
«Java Class»
MainActivity
viewPager: ViewPager
tabsPagerAdapter: TabsPagerAdapter
actionBar: ActionBar
mContext: Context
bw: BeaconWhisperer
MainActivity()
onBackPressed():void
onCreate(Bundle):void
onCreateOptionsMenu(Menu):boolean
onTabReselected(Tab,FragmentTransaction):void
onTabSelected(Tab,FragmentTransaction):void
onTabUnselected(Tab,FragmentTransaction):void
onOptionsItemSelected(MenuItem):boolean
openDemo():void
openSettings():void
checkIfLogOut():void
getContext():Context
onActivityResult(int,int,Intent):void
«Java Class»
LoginActivity
linfo: LoginInfo
u: EditText
p: EditText
cb: CheckBox
LoginActivity()
onCreate(Bundle):void
onCreateOptionsMenu(Menu):boolean
login():void
onButtonClicked(View):void
validate():void
onCheckboxClicked(View):voic
«Java Class»
LoginTask
LoginTask()
doInBackground(Void[]):Boolean
onPostExecute(Boolean):void
«Java Class»
PersonActivity
user: User
location: String
nameView: TextView
roleView: TextView
posView: TextView
imgView: ImageView
toggleFav: Button
searchButton: Button
favorite: boolean
PersonActivity()
onCreate(Bundle):void
onCreateOptionsMenu(Menu):boolean
locateUser(View):voic
displayUser():void
listToString(ArrayList<String>,String):String
toggleButton(View):void
isFavorite():boolean
callUser(View):voic
Hämta info om person med ServerCon
«Java Class»
FavoritesFragment
users: List<User>
FavoritesFragment()
TEST__createUserList():List<User>
onActivityCreated(Bundle):void
onCreateView(LayoutInflater,ViewGroup,Bundle):View
onListItemClick(ListView,View,int,long):voic
«Java Class»
LogbookFragment
LogbookFragment()
onCreateView(LayoutInflater,ViewGroup,Bundle):View
onListItemClick(ListView,View,int,long):voic
onActivityCreated(Bundle):void
TEST__createLogEntryList():ArrayList<LogEntry>
TEST__createUserList():List<User>
Hämta loggbok med ServerCon

En annan väldigt viktig klass är BeaconWhisperer, vilken hanterar kommunikationen med beacons. BeaconWhisperer ansvarar för lagring av närmaste beacon och att anropa metoder i externa bibliotek för att starta sökning efter beacons. BeaconWhisperer ska även innehålla funktionalitet för att med jämna mellanrum hämta information om batterinivån hos beacons, för att sedan, med hjälp av ServerCon, rapportera dessa data till servern.



4.2.3 Coupling och cohesion

I skapandet av arkitekturen togs vissa designprinciper i åtanke. Ett mål har exempelvis varit att alla klasser ska uppfylla en funktion vardera, även om vissa undantag görs. I arkitekturen finns två klasser som hanterar kommunikation mellan klient och server, nämligen ServerCon hos klienten och Listener hos servern. Dessa klasser är designade med avsikten att ha relativt låg coupling, det vill säga att funktionaliteten av klasserna ska vara så opåverkade av varandras framtida förändring som möjligt, och hög cohesion, det vill säga att varje komponents inre funktionalitet är sammanhängande.

I Listener skapas diverse Procedures som ska hantera och skicka svar på förfrågningar från klienter. En tanke med Listener är att den skall fungera oberoende av dessa Procedures inre maskinerier. Listeners enda kommunikation med Procedures består av skapandet av nya objekt och anrop till metoden startProcedure. Hur dessa Procedures fungerar är inte av intresse för Listener; Listener är bara intresserad av att de startar vid rätt tillfälle. Detta tillvägagångssätt har

inte nödvändigtvis lägst möjliga coupling, men målet med arkitekturen har varit att minimera coupling i så stor utsträckning som möjligt.

Lägre coupling hade kunnat uppnås med användning av Mediator-mönstret, där kommunikation mellan klasser sker genom ett meddelandesystem [2]. Med ett sådant system hade det inte funnits något behov av Listener att direkt anropa startProcedure, utan den hade istället utlösts genom en "förmedlare" (mediator). Mediator är ett mer framtidssäkert sätt att hantera kommunikation mellan klasser. Om Procedure, som den används enligt vår arkitektur, i framtiden ändras till den grad att startProcedure() inte längre existerar, kommer Listener att sluta fungera. Med Mediator-mönstret hade denna kommunikation från Listeners sida då varit oförändrad, trots ändringarna i Procedure.

En direkt nackdel med Mediator-mönstret är dock att användandet av den innebär större komplexitet för arkitekturen med potentiellt fler komponenter och större arbetsbörda för utförandet av projektet. Valet att inte använda detta designmönster motiveras av att vi utvecklar en prototyp och inte en färdig produkt, så det är viktigare att vi får funktionerna att fungera än att framtidssäkra produkten.

Hög cohesion innebär ur ett objektorienterat perspektiv att metoderna som utgör en klass har en eller flera gemensamma funktionella egenskaper. Exempel på klasser med låg cohesion är så kallade utility-klasser, som innehåller vanligt förekommande metoder vilka utför helt orelaterade funktionaliteter. Ett exempel på en typisk utility-klass är java.util.Collections vilken består av diverse statiska metoder. Java.util.Collections har låg cohesion eftersom dess metoder inte är funktionellt sammanhängande; de existerar för helt olika syften.

I arkitekturen finns ett antal klasser med hög cohesion som existerar för att behandla en viss typ av data, som exempelvis DBConn som hanterar databasoperationer för serverkomponenter och FileManager som hanterar filoperationer för klientkomponenter. Dessa klasser designades med avsikten att varje metod ska utföra en funktionalitet, och att denna funktionalitet ska vara relevant för syftet med klassen. De klasser som har något mindre cohesion är de klasser vilkas uppgift är att behandla större mängder data i flera steg. Ett exempel är Listener, som i sekvens tar emot en anslutning från en klient, skapar en ny tråd för att hantera anslutningen, parsar den data som tas emot och skickar vidare parsad data för behandling. Högre cohesion hade i denna klass kunnat uppnås genom att bryta ut viss funktionalitet till andra klasser, som exempelvis parsing. Varför detta inte gjordes beror, precis som med valet att inte ha Mediator-mönstret, på

att slutprodukten enbart är tänkt att vara en prototyp. Fler komponenter skulle kunna öka arkitekturens komplexitet, vilket inte är önskvärt.

4.2.4 Protokoll

Data till servern skickas som JSON-strängar. Fältet action måste inkluderas i JSON-strängen. Exempel:

```
{
     “action”:”REQUEST_LOGIN”,
     …
}
```

4.2.4.1 Möjliga värden på action

Möjliga värden på fältet action är:

1. REQUEST_LOGIN
2. REQUEST_LOGOUT
3. REQUEST_POSITION
4. POSITION_RESPONSE
5. REQUEST_LOG
6. REQUEST_SEARCH_RESULTS
7. REQUEST_FAVORITES
8. REQUEST_FAVORITES_REMOVAL

Nedan förklaras varje värde för sig.

REQUEST_LOGIN

Logga in användaren på servern. Kräver följande fält tillsammans med action:

```
“uid”:”myUsername”
“password”:1337
“registration_id”:”34734834938473487393”
```

REQUEST_LOGOUT

Logga ut från servern genom att sätta token till null. Kräver följande fält tillsammans med action:

```
“uid”:”myUsername”
“token”:”myToken”
```

REQUEST_POSITION

Begär en annan användares position. Kräver följande fält tillsammans med action:

```
“requester”:”myUsername”
“requested”:”usernameOfRequestedUser”
“token”:”myToken”
```

POSITION_RESPONSE

Skicka användarens position till servern. Detta sker automatiskt efter att applikationen fått en push-notifikation via GCM. Kräver följande fält tillsammans med action:

```
“requester”:”usernameOfUserWhoMadeTheRequest” (kommer skickas till appen via GCM)
“requested”:”myUsername”
“major”:12313543
“minor”:342234
```

REQUEST_LOG

Begär loggen med positionsförfrågningar som gjorts på användaren. Kräver följande fält tillsammans med action:

```
“uid”:”myUsername”
“token”:”myToken”
```

REQUEST_SEARCH_RESULTS

Begär sökresultat med hjälp av en söksträng. Kräver följande fält tillsammans med action:

```
“token”:”myToken”
“uid”:”myUsername”
“pattern”:”search_pattern”
```

REQUEST_FAVORITES

Begär användarens favoriter. Kräver följande fält tillsammans med action:

```
“token”:”myToken”
“uid”:”myUsername”
```

REQUEST_FAVORITES_REMOVAL

Ta bort valda favoriter. Kräver följande fält och array tillsammans med action::

```
“uid”:”myUsername”
“token”:”myToken”
“removals”:[
            {“name”:”A”},
            {“name”:”B”},
            …
           ]
```

A, B… är namn på favoriter som skall tas bort.

4.2.4.2 Inloggning

1. Klient ansluter till server.

2. Klient skickar inloggningsuppgifter:

```
{
     “action”:”REQUEST_LOGIN”
     “uid”:”bastian”
     “password”:4216
     “registration_id”:”4589340394830493849”
}
```

3. Servern svarar med token eller en tom sträng om inloggningen misslyckades.

4.2.4.3 Utloggning

1. Klient ansluter till server.

2. Klient skickar autentiseringsuppgifter:

```
{
     “action”:”REQUEST_LOGOUT”
     “token”:”4i4jwer90vsd9j345iop0dwdoq3p5o4i903i4k3jr3i”
}
```

4.2.4.4 Begär position

1. Klient ansluter till server.

2. Klient skickar positionsdata:

```
{
     “action”:”REQUEST_POSITION”
     “requested”:”tom”
     “requester”:”bastian”

“token”:”34jerk23jkjefk1235olfkekl4klwel23454kj45lk4lwek5r34l534kjk454lkj”
}
```

3. Servern levererar svar (rumsnummer) till klient om klient ej gjort timeout och stängt anslutningen:

```
{
      “room_number”:”Mordor”
}
```

4.2.4.5 Skicka position

1. Klienten får en callback från GCM med payload:

```
{
     “requester”:”bastian”
     “token”:”myToken”
}
```

2. Klienten skickar positionsdata och position:

```
{
     “action”:”POSITION_RESPONSE”
     “requester”:”bastian”
    “requested”:”tom”
     “major”:452343
     “minor”:2403834
}
```

3. Servern levererar svar (rumsnummer) till klient om klient ej gjort timeout och stängt anslutningen:

```
{
      “room_number”:”Mordor”
}
```

4.2.4.6 Begär logg

1. Klient ansluter till server.

2. Klient skickar autentiseringsuppgifter:

```
{
     “uid”:”bastian”
     “token”:”4fk340vkweklrtasdolmt5450fvj345r0vuj4je945krjekk4”
     “action”:”REQUEST_LOG”
}
```

3. Server svarar med logg i form av LogEntryList-objekt.

4.2.4.7 Begär favoriter

1. Klient ansluter till server.

2. Klient skickar autentiseringsuppgifter:

```
{
    “uid”:”bastian”
    “token”:”4fk340vkweklrtasdolmt5450fvj345r0vuj4je945krjekk4”
```

```
    “action:”REQUEST_FAVORITES”
}
```

3. Servern svarar med favoriter i form av UserList-objekt.

4.2.4.8 Begär borttagning av favoriter

1. Klienten ansluter till server.

2. Klienten skickar autentiseringsuppgifter och en lista med favoriter som skall tas bort.

```
{
    “uid”:”bastian”
    “token”:”4fk340vkweklrtasdolmt5450fvj345r0vuj4je945krjekk4”
    “removals”:[  {“name”:”elias”},
            {“name”:”joakim”},
             …
                 ]
}
```

4.2.4.9 Begär sökresultat

1. Klient ansluter till server.

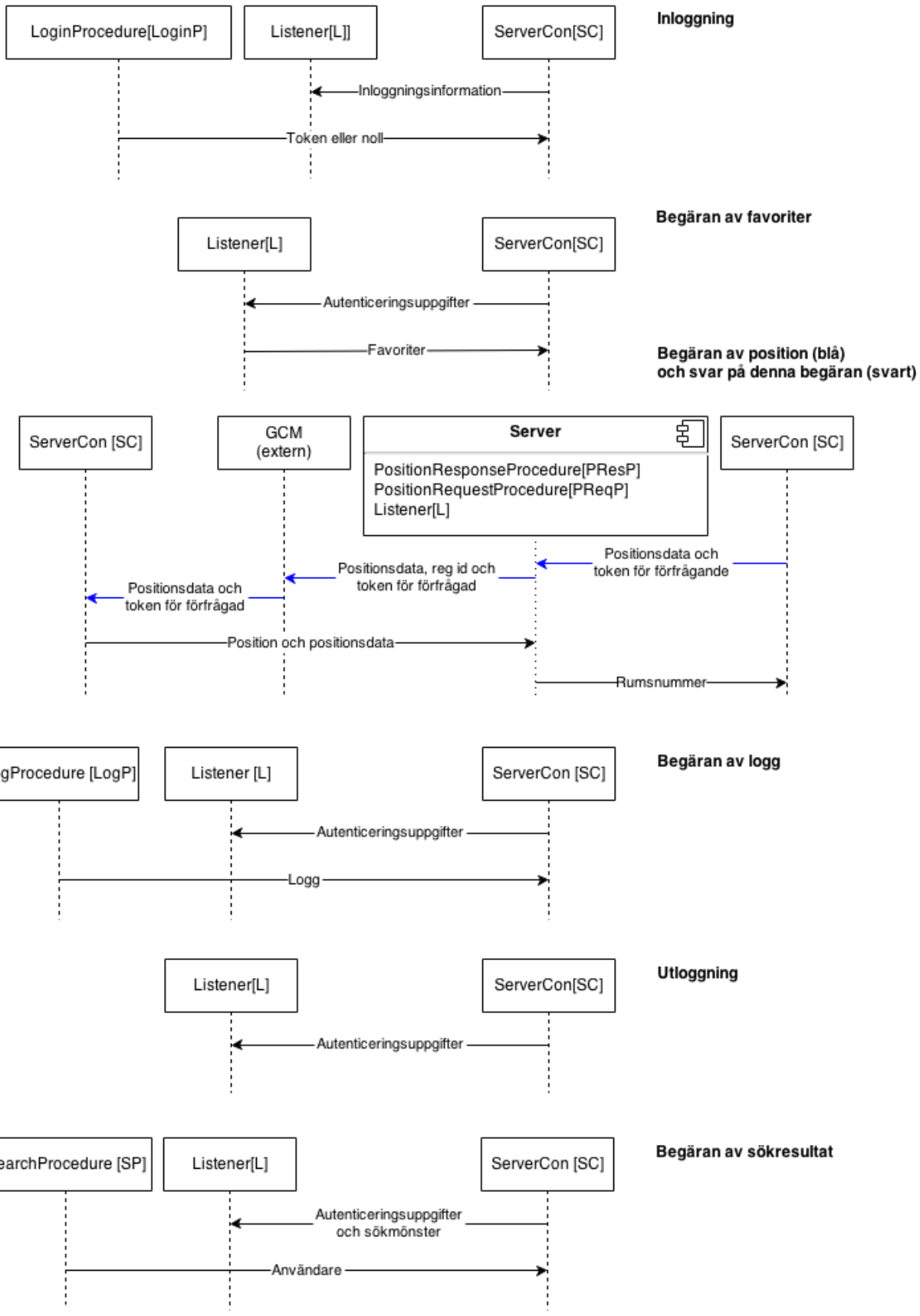
2. Klient skickar autentiseringsuppgifter och sökmönster:

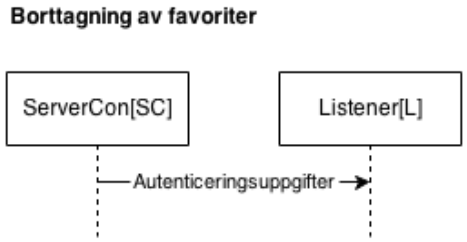
```
{
    “uid”:”bastian”
    “token”:”4fk340vkweklrtasdolmt5450fvj345r0vuj4je945krjekk4”
    “action:”REQUEST_SEARCH_RESULTS”
    “pattern”:”tom”
}
```

3. Server svarar med användare i form av UserList-objekt.

4.2.4.10 Protokollet i grafisk form

Nedan illustreras protokollets alla delar i ett UML-diagram.


Inloggning
LoginProcedure[LoginP]
Listener[L]]
ServerCon[SC]
Inloggningsinformation
Token eller noll
Begäran av favoriter
Listener[L]
ServerCon[SC]
Autenticeringsuppgifter
Favoriter
Begäran av position (blå)
och svar på denna begäran (svart)
ServerCon [SC]
GCM
(extern)
Server
PositionResponseProcedure[PResP]
PositionRequestProcedure[PReqP]
Listener[L]
ServerCon [SC]
Positionsdata och
token för förfrågande
Positionsdata, reg id och
token för förfrågad
Positionsdata och
token för förfrågad
Position och positionsdata
Rumsnummer
Begäran av logg
LogProcedure [LogP]
Listener [L]
ServerCon [SC]
Autenticeringsuppgifter
Logg
Utloggning
Listener[L]
ServerCon[SC]
Autenticeringsuppgifter
Begäran av sökresultat
SearchProcedure [SP]
Listener[L]
ServerCon [SC]
Autenticeringsuppgifter
och sökmönster
Användare



4.2.5 Databas

För att hantera användare, användarinformation, beacons, rum, avdelningar och andra data, använder vi oss av en MySQL-databas. För oss stod valet från början mellan MySQL och PostgreSQL. Det som talade för PostgreSQL var att vi inom gruppen hade tidigare erfarenheter av PostgreSQL, men valet föll slutligen på MySQL eftersom den överlag är ansedd att vara snabbare än PostgreSQL. I detta projekt är prestandan viktigare än den utökade funktionalitet som PostgreSQL erbjuder. MySQL är även vida spritt och uppfattas som lite av en branschstandard när det gäller databaser som bygger på öppen källkod. Även MySQL Workbench vägde för i vårt val av databas eftersom vi upplever den som mer robust och användarvänlig än PostgreSQL:s motsvarighet pgAdmin III.

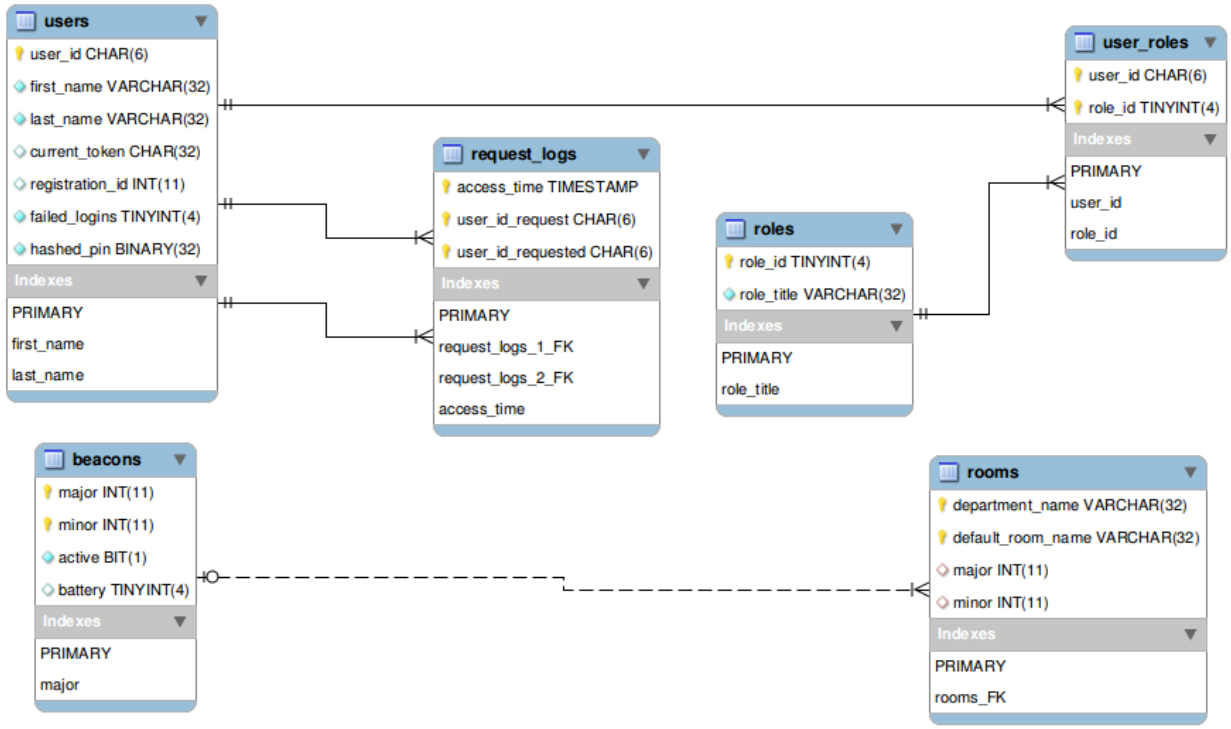
Vi har lagt in checks för vissa av attributen för att begränsa vilka värden de kan anta. Dessvärre stödjer inte MySQL detta i dagsläget, men vi har lagt in det i hopp om att det ska stödjas i framtida releaser. [6] För att försäkra oss om att felaktig data inte lagras i databasen har vi istället lagt kontrollerna i serverns Java-applikation.

4.2.5.1 Designprocessen för databastabellerna

Att designa databasen har varit ett omfattande arbete med ett flertal iterationer innan vi var framme vid en välfungerande databasmodell. Vi använde oss av en top-down designmetod där vi började med en övergripande design med ett fåtal tabeller. Dessa tabeller bröts sedan ned i fler och fler tabeller för varje iteration när vi upptäckte att vi behövde mer funktionalitet.

Nedan beskrivs vilka iterationer databasen genomgått.

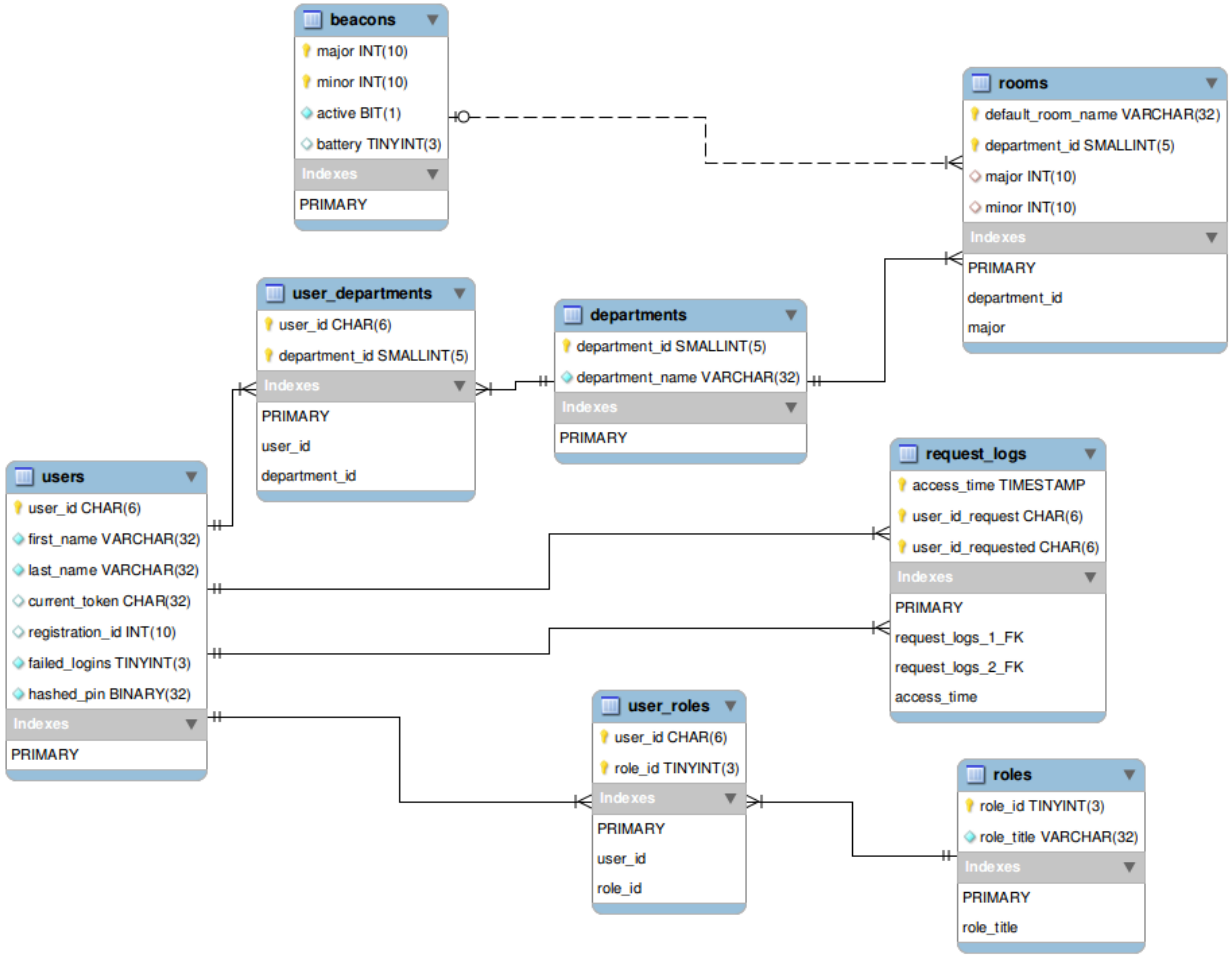
Version 0.1



Detta är den första versionen av databasmodellen. Endast den mest grundläggande funktionaliteten är implementerad. Här var inte alla datatyper för de olika attributen bestämda ännu och flertalet kom att ändras under arbetets gång. users innehåller här grundläggande information om användaren som exempelvis namn, token och lösenord. request_logs innehåller en tidsstämpel och kopplingar till två olika användare där den ena är den som begärt positionen på den andra. roles innehåller namnen på de roller som finns och ett unikt ID för varje roll. user_roles binder samman users med roles via ett user_id och ett role_id. beacons innehåller information om major och minor för beacons. rooms innehåller namn för rum och vilken avdelning de tillhör. I denna version binds rooms till ingen eller en beacon.

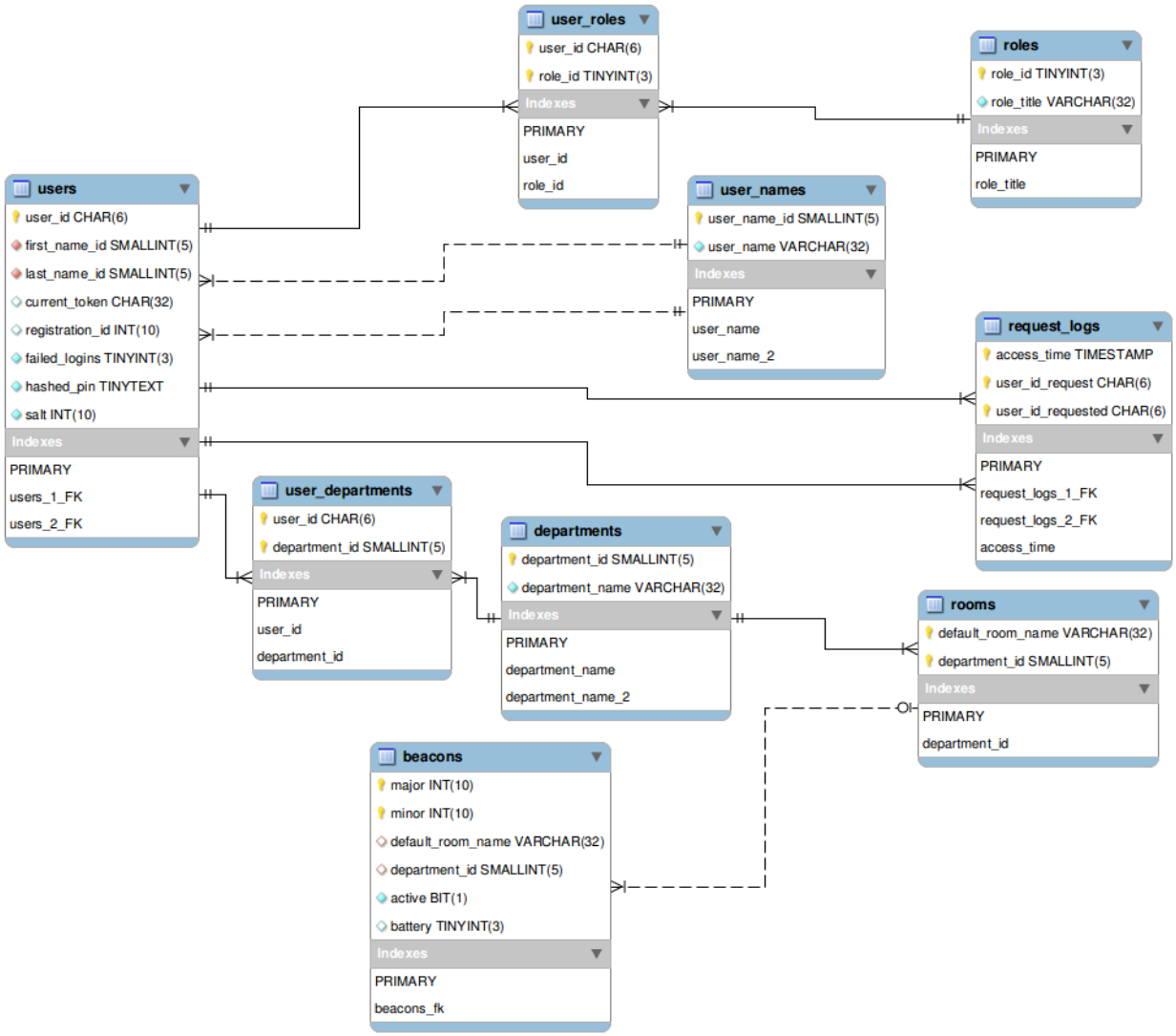
Av dessa tabeller var det bara request_logs som inte ändrades mer medan roles och user_roles fick minimala ändringar. Övriga tabeller gjordes om i större utsträckning under arbetets gång.

Version 0.2



Till denna version har vi lagt till departments för att kunna koppla rum och användare till avdelningar. För att koppla samma användare till flera avdelningar används tabellen user_departments. Flera av de numeriska datatyperna har ändrats till unsigned för att det inte finns något behov av negativa värden.

Version 0.3

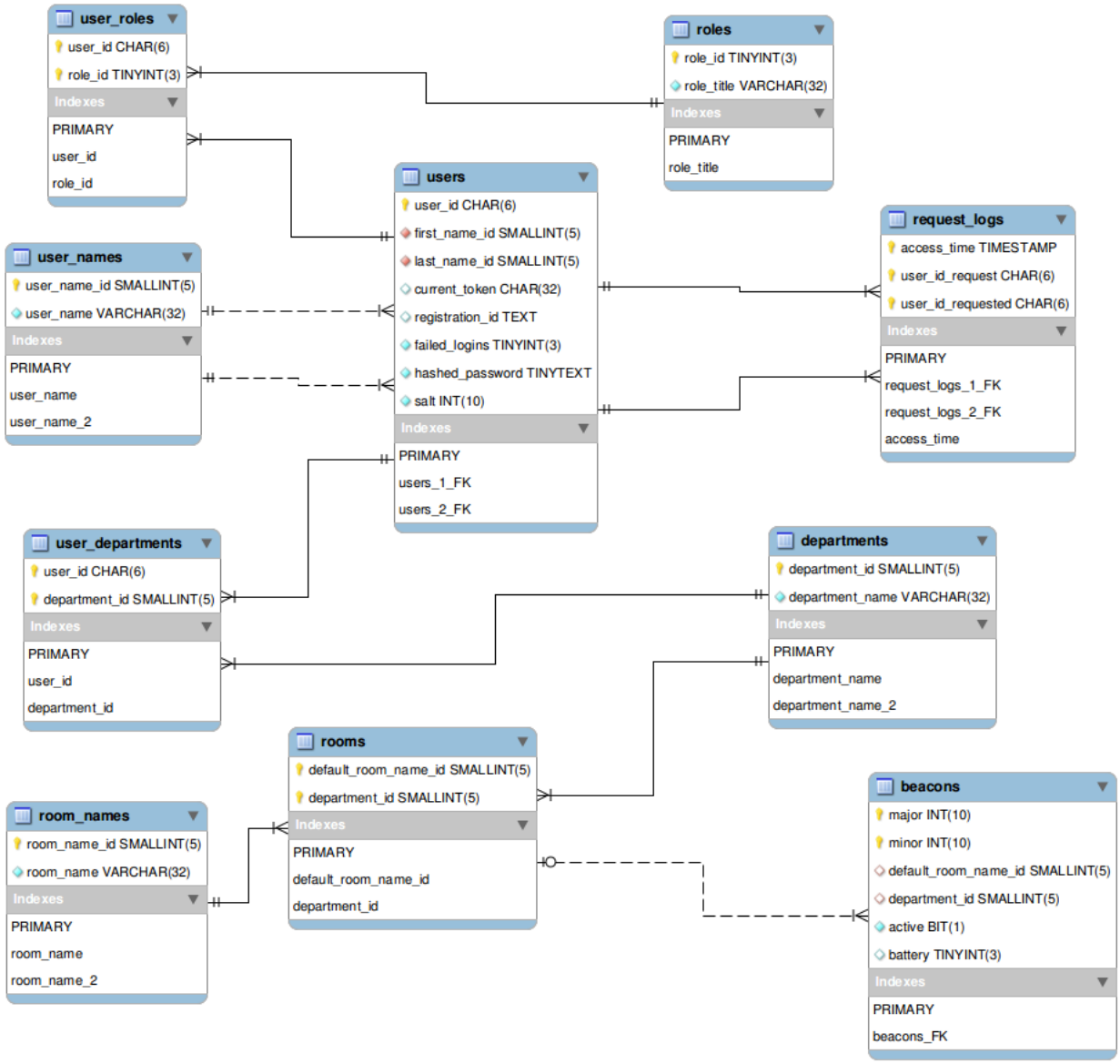


Kopplingen mellan beacon och rooms var tidigare felaktig; den gick i fel riktning. Tidigare kopplade ett rum till ingen eller en beacon, vilket innebar att det endast var möjligt att koppla en beacon till varje rum. Tanken är att det ska vara möjligt att koppla flera beacons till ett rum. Detta gav den omvända kopplingen, att en beacon kan kopplas till ingen eller ett rum.

För att effektivisera databasen genom att undvika att användarnas namn dubbellagras om flera användare har samma förnamn eller efternamn skapades tabellen user_names. I denna lagras alla namn endast en gång och binds till en surrogatnyckel som lagras i users för att binda ihop user_names och users. Förnamn och efternamn lagras i samma kolumn i user_names för att ytterligare spara utrymme i databasen om det finns dubbletter mellan förnamn och efternamnen.

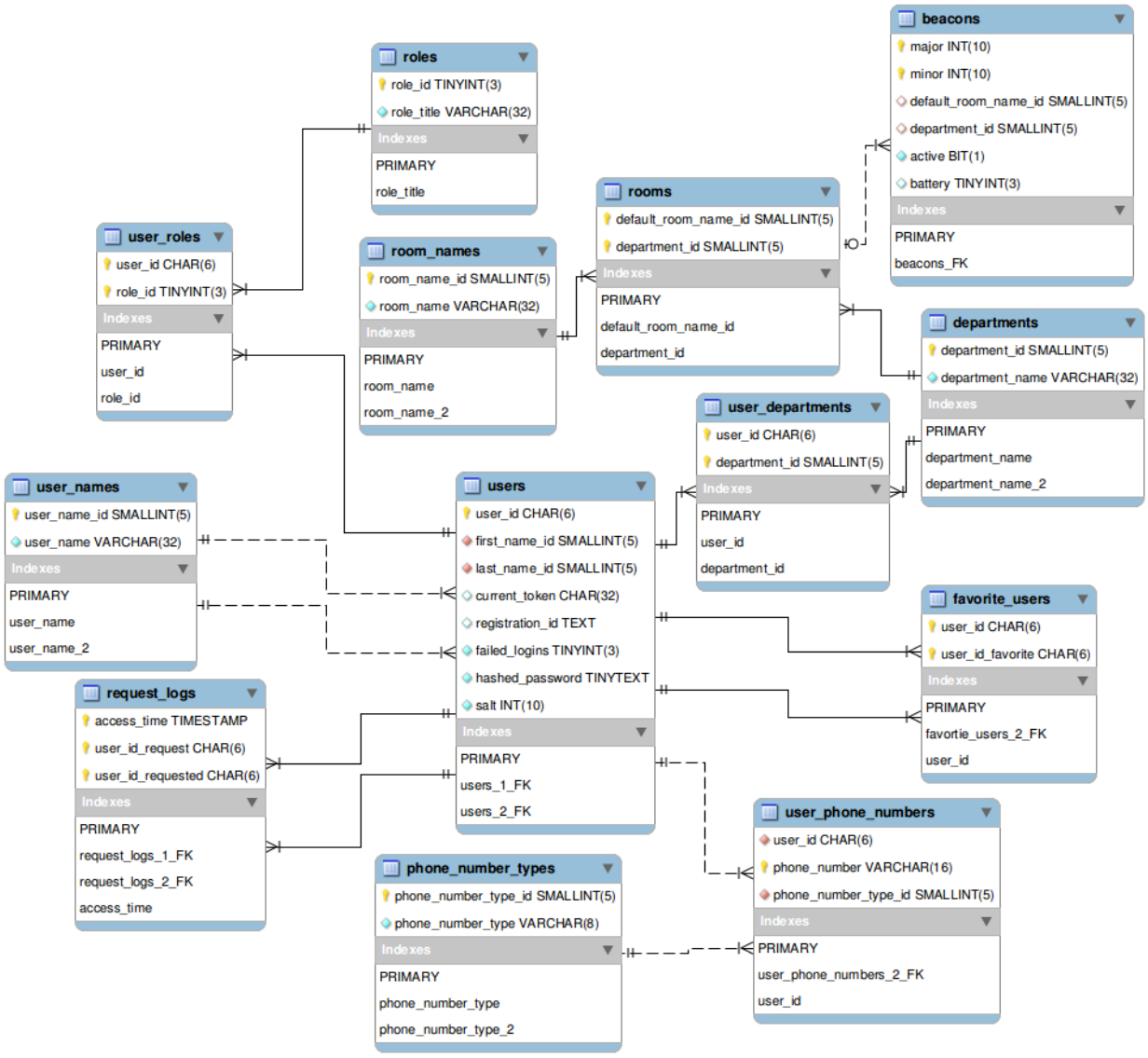
I users finns i denna version attributet salt som innehåller en slumpmässigt genererad int. Denna kommer att konkateneras till lösenordet innan det hashas för att öka säkerheten.

Version 0.4



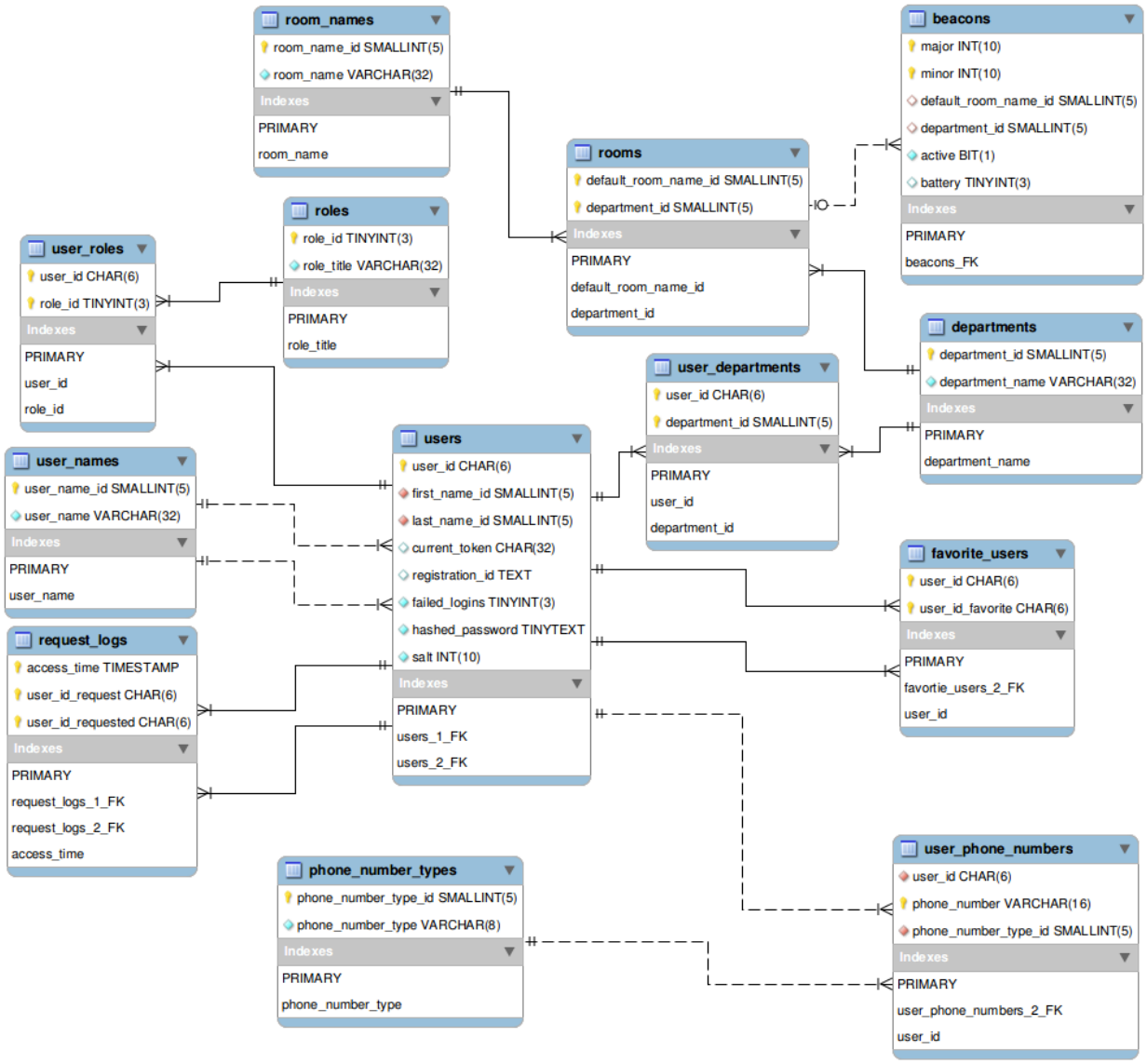
För att undvika dubbellagring av rumsnamnen flyttades dessa till en egen tabell, eftersom det är mycket troligt att flera rum heter samma sak fast de ligger på olika avdelningar. Detta förbereder även för att i framtiden kunna bygga ut databasen med alternativa rumsnamn.

Version 0.5



Denna version innefattar även tabellerna favorite_users, user_phone_numbers och phone_number_types. Tabellen favorite_users används för att spara användarens favoriter och user_phone_numbers tillsammans med phone_number_types används för att spara användarens telefonnummer.

Version 1.0



Detta är den slutgiltiga versionen av databasen. I denna version är onödiga dubbletter av index borttagna.

4.2.5.2 Detaljerad beskrivning av databastabellerna

Nedan beskrivs samtliga tabeller som genom den iterativa processen valts ut för att kunna lagra all information som behövs om användare, utan att lagra samma information flera gånger då detta kan leda till uppdateringsanomalier och liknande. Vi använder oss av foreign key-restriktioner för att försäkra att inga tabeller refererar till användare, roller eller avdelningar som inte finns. Alla index har valts ut för att optimera de sökningar som är mest sannolika.

Figuren ovan (i Kapitel 4.2.5.1) illustrerar ett EER-diagram med alla tabeller.

users

Denna tabell har information om användarna. User_id utgör nyckeln i tabellen och är användarens unika användarnamn som används för att logga in i applikationen. First_name_id och last_name_id är vardera en foreign key till tabellen user_names. Current_token är en slumpmässigt genererad token som används för att identifiera användaren utan att behöva skicka user_id och password vid kommunikationen mellan klient och server, och den uppdateras varje gång användaren loggar in. Failed_logins är antalet misslyckade inloggningsförsök som användaren genomfört sedan den senaste lyckade inloggningen. Denna nollställs vid varje lyckad inloggning, men efter tre misslyckade inloggningar spärras användarens konto. Hashed_password innehåller en saltad och hashad version av användarens lösenord. Salt är ett slumpmässigt genererat heltal som konkateneras till användarens lösenord innan det hashas.

Users är den centrala tabellen i databasen och knyter ihop resterande tabeller i systemet. Databasens huvudsyfte är ju att hantera information rörande användarna.

Attribut

- user_id CHAR(6) NOT NULL
- first_name_id SMALLINT UNSIGNED NOT NULL
- last_name_id SMALLINT UNSIGNED NOT NULL
- current_token CHAR(32)
- registration_id TEXT
- failed_logins TINYINT UNSIGNED NOT NULL
- hashed_password TINYTEXT NOT NULL
- salt INT UNSIGNED NOT NULL

Regler

- CONSTRAINT users_PK PRIMARY KEY (user_id)
- CONSTRAINT users_1_FK FOREIGN KEY (first_name_id) REFERENCES user_names (user_name_id)
- CONSTRAINT users_2_FK FOREIGN KEY (last_name_id) REFERENCES user_names (user_name_id)
- CONSTRAINT failed_logins_CHK CHECK (failed_logins < 3)
  Denna regel finns för att förhindra att obehöriga försöker sig på en brute force-attack för att komma åt en användares lösenord.

Index

- INDEX(user_id)
- INDEX(first_name_id)
- INDEX(last_name_id)

user_names

Denna tabell innehåller samtliga namn under attributet user_name och en surrogatnyckel user_name_id för att binda namn till användare i user-tabellen. Denna tabell syftar endast till att öka prestandan i databasen genom att minska eventuell dubbellagring av användarnas namn. Tabellen lagrar både förnamn och efternamn i samma kolumn för att minska risken för dubbellagring av namn om något förnamn stämmer överens med något efternamn.

Attribut

- user_name_id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT
- user_name VARCHAR(32) NOT NULL UNIQUE

Regler

- CONSTRAINT user_names_PK PRIMARY KEY (user_name_id),

Index

- INDEX (user_name_id)
- INDEX (user_name)

roles

Denna tabell lagrar information om vilka olika roller i attributet role_title som användare kan tilldelas och har nyckeln role_id för att binda roller till tabellen user_roles. Denna tabells syfte är att säkerställa att en användare endast kan tilldelas definierade roller.

Attribut

- role_id TINYINT UNSIGNED NOT NULL AUTO_INCREMENT
- role_title VARCHAR(32) NOT NULL UNIQUE

Regler

- CONSTRAINT roles_PK PRIMARY KEY (role_id)

Index

- INDEX (role_id)
- INDEX (role_title)

user_roles

Denna tabell binder ihop användare via user_id med de roller användarna tilldelats via role_id. Denna tabell behövs för att en användare ska kunna tilldelas flera roller.

Attribut

- user_id CHAR(6) NOT NULL
- role_id TINYINT UNSIGNED NOT NULL

Regler

- CONSTRAINT user_roles_PK PRIMARY KEY (user_id, role_id)

- CONSTRAINT user_roles_1_FK FOREIGN KEY (user_id) REFERENCES users (user_id)
- CONSTRAINT user_roles_2_FK FOREIGN KEY (role_id) REFERENCES roles (role_id)

Index

- INDEX (user_id, role_id)
- INDEX (user_id)
- INDEX (role_id)

departments

Denna tabell har information om vilka olika avdelningar som en användare kan arbeta på och vilka avdelningar ett rum kan tillhöra. Avdelningarnas namn sparas under department_name. Department_id är en surrogatnyckel som används för att binda ihop departments med user_departments och rooms. Denna tabells syfte är att minska dubbellagring av avdelningsnamn och säkerställa att användare och rum endast kan bindas till definierade avdelningar.

Attribut

- department_id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT
- department_name VARCHAR(32) NOT NULL UNIQUE

Regler

- CONSTRAINT departments_PK PRIMARY KEY (department_id)

Index

- INDEX (department_id)
- INDEX (department_name)

user_departments

Denna tabell binder ihop användare med avdelningarna de arbetar på. Denna tabell behövs för att en användare ska kunna bindas till flera avdelningar. Attributet user_id binder tabellen till användare, och department_id binder tabellen till avdelningar.

Attribut

- user_id CHAR(6) NOT NULL
- department_id SMALLINT UNSIGNED NOT NULL

Regler

- CONSTRAINT user_department_PK PRIMARY KEY (user_id, department_id)
- CONSTRAINT user_department_1_FK FOREIGN KEY (user_id) REFERENCES users (user_id)
- CONSTRAINT user_department_2_FK FOREIGN KEY (department_id) REFERENCES departments (department_id)

Index

- INDEX (user_id, department_id)
- INDEX (user_id)
- INDEX (department_id)

favorite_users

Denna tabell innehåller referenser till varje användares sparade favoriter. Attributet user_id knyter ihop en användare med dennes favoriter som refereras till med attributet user_id_favorite.

Attribut

- user_id CHAR(6) NOT NULL
- user_id_favorite CHAR(6) NOT NULL

Regler

- CONSTRAINT favorite_users_PK PRIMARY KEY (user_id, user_id_favorite)
- CONSTRAINT favortie_users_1_FK FOREIGN KEY (user_id) REFERENCES users (user_id)
- CONSTRAINT favortie_users_2_FK FOREIGN KEY (user_id_favorite) REFERENCES users (user_id)

Index

- INDEX (user_id, user_id_favorite)
- INDEX (user_id_favorite)
- INDEX (user_id)

request_logs

Denna tabell har information om vilka användare som sökt efter andra användare och när sökningarna skett. Access_time uppdateras automatiskt när en ny rad läggs in i databasen. User_id_request refererar till den användare som efterfrågat en användares position. Den efterfrågade användaren refereras till med user_id_requested.

Attribut

- access_time TIMESTAMP NOT NULL ON UPDATE CURRENT_TIMESTAMP
- user_id_request CHAR(6)
- user_id_requested CHAR(6)

Regler

- CONSTRAINT request_logs_PK PRIMARY KEY (access_time, user_id_request, user_id_requested)
- CONSTRAINT request_logs_1_FK FOREIGN KEY (user_id_request) REFERENCES users (user_id)
- CONSTRAINT request_logs_2_FK FOREIGN KEY (user_id_requested) REFERENCES users (user_id)
- CONSTRAINT user_id_CHK CHECK (user_id_request <> user_id_requested) Denna regel förhindrar att en användares sökning efter sig själv loggas.

Index

- INDEX (access_time, user_id_request, user_id_requested)
- INDEX (user_id_request)
- INDEX (user_id_requested)
- INDEX (access_time, user_id_requested)

user_phone_numbers

Denna tabell binder användare med deras telefonnummer, och behövs för att en användare ska kunna knytas till flera telefonnummer samtidigt. User_id refererar till en användare, phone_number refererar till användarens telefonnummer och phone_number_type_id refererar till tabellen phone_number_type.

Attribut

- user_id CHAR(6) NOT NULL
- phone_number VARCHAR(16) NOT NULL
- phone_number_type_id SMALLINT UNSIGNED NOT NULL

Regler

- CONSTRAINT user_phone_numbers_PK PRIMARY KEY (phone_number)
- CONSTRAINT user_phone_numbers_1_FK FOREIGN KEY (user_id) REFERENCES users (user_id)
- CONSTRAINT user_phone_numbers_2_FK FOREIGN KEY (phone_number_type_id) REFERENCES phone_number_types (phone_number_type_id)

Index

- INDEX (phone_number)
- INDEX (phone_number_type_id)
- INDEX (user_id)

phone_number_types

Denna tabell innehåller de typer av telefonnummer som är möjliga att spara ett nummer som. Surrogatnyckeln phone_number_type_id används för att binda typen till telefonnummer i tabellen user_phone_numbers. Attributet phone_number_type lagrar typen. Denna tabells syfte är att säkerställa att telefonnummer endast binds till definierade typer.

Attribut

- phone_number_type_id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT
- phone_number_type VARCHAR(8) NOT NULL UNIQUE

Regler

- CONSTRAINT phone_number_types_PK PRIMARY KEY (phone_number_type_id)

Index

- INDEX (phone_number_type_id)
- INDEX (phone_number_type)

beacons

Denna tabell håller information om beacons. Major och minor utgör tillsammans den unika nyckeln som gör det möjligt att identifiera en specifik beacon. Default_room_name_id och department_id används för att binda en beacon till ett rum, och dessa attribut får anta NULL eftersom det ska vara möjligt att lägga till beacons i databasen utan att de är uppsatta och bundna till ett specifikt rum. Attributet active är till för att kunna avgöra om en viss beacon är uppsatt och används eller inte. Attributet battery kan användas för att lagra batteristatusen.

Attribut

- major INT UNSIGNED NOT NULL
- minor INT UNSIGNED NOT NULL
- default_room_name_id SMALLINT UNSIGNED
- department_id SMALLINT UNSIGNED
- active bit NOT NULL
- battery TINYINT UNSIGNED

Regler

- CONSTRAINT beacons_PK PRIMARY KEY (major, minor),
- CONSTRAINT beacons_FK FOREIGN KEY (default_room_name_id, department_id) REFERENCES rooms (default_room_name_id, department_id),
- CONSTRAINT battery_CHK CHECK (battery <= 100 OR battery IS NULL)
  Denna regel förhindrar att ett batterivärde som ligger utanför intervallet 0-100 % sparas.

Index

- INDEX (major, minor)
- INDEX (default_room_name_id, department_id)

rooms

Denna tabell har information om de rum som finns och binder ihop dem med beacons. Default_room_name_id binder rooms till room_names. Department_id binder rooms till departments.

Attribut

- default_room_name_id SMALLINT UNSIGNED NOT NULL
- department_id SMALLINT UNSIGNED NOT NULL

Regler

- CONSTRAINT rooms_PK PRIMARY KEY (default_room_name_id, department_id)
- CONSTRAINT rooms_1_FK FOREIGN KEY (default_room_name_id) REFERENCES room_names (room_name_id)
- CONSTRAINT rooms_2_FK FOREIGN KEY (department_id) REFERENCES departments (department_id)

Index

- INDEX (default_room_name_id, department_id)
- INDEX (default_room_name_id)
- INDEX (department_id)

room_names

Denna tabell innehåller alla rumsnamn och en surrogatnyckel room_name_id för att binda namn i room_name till rum i tabellen room. På så sätt undviks dubbellagring av rumsnamn eftersom samma namn kan förekomma på flera rum i olika avdelningar.

Attribut

- room_name_id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT
- room_name VARCHAR(32) NOT NULL UNIQUE

Regler

- CONSTRAINT room_names_PK PRIMARY KEY (room_name_id)

Index

- INDEX (room_name_id)
- INDEX (room_name)

5 Component Description

För att beskriva metoder använder vi oss av formatet *namn(parametertyper) : returtyp*. Exempelvis betyder `writeFavorites(UserList) : void` att metoden writeFavorites tar en parameter av typen UserList, och returnerar ingenting (void).

För att beskriva fält använder vi oss av formatet *namn : typ*. Exempelvis betyder `errorMessage : String` att fältet errorMessage är av typen String.

5.1 [FF]

FavoritesFile.

5.1.1. Type

Fil.

5.1.2. Purpose

Syftet med filen är att kunna spara sina favoritanvändare. Informationen ska finnas kvar även när applikationen har stängts ner.

Filen möjliggör en utvidgning av URD-krav 3.1.4 i form av att tillåta sökning på favoritanvändare. Dessutom är filen relevant för att snabbt kunna hitta användare som man troligen söker på och att slippa distraktionen med att söka på ett namn varje gång. Detta relaterar till URD-kraven 3.2.6 och 3.2.7.

5.1.3. Function

Filen är en behållare av nycklar som kan användas till att identifiera ens favoritanvändare.

5.1.4. Subordinates

N/A

5.1.5. Dependencies

N/A

5.1.6. Interfaces

PersonActivity[PA] kan läsa och skriva till filen genom FileManager[FM]. Läsning motsvarar att man hämtar informationen i filen för att kunna lista favoritanvändare i applikationen. Skrivning motsvarar tilläggning eller borttagning av favoritanvändare. När man lyckas logga in i applikationen kommer servern att skicka ner ens favoriter, vilka sedan sparas lokalt i filen för att minska på kommunikationen med servern.

5.1.7. Resources

N/A

5.1.8. References

N/A

5.1.9. Processing
N/A

5.1.10. Data
Filen lagrar nycklar på samtliga favoritanvändare för respektive användare. Nycklarna är favoritanvändarnas användarnamn.

5.2 [LD]
LoginDetails.

5.2.1. Type
Fil.

5.2.2. Purpose
För att möjliggöra en automatisk inloggning krävs det att en token från senaste lyckade inloggningen lagras någonstans i kombination med användarnamn.

Filen är relevant för URD-krav 3.2.7 eftersom det vore ett distraktionsmoment att ständigt behöva skriva in sina inloggningsuppgifter.

5.2.3. Function
Filen är en behållare av inloggningsdata.

5.2.4. Subordinates
N/A

5.2.5. Dependencies
N/A

5.2.6. Interfaces
LoginActivity[LA] läser och skriver till filen genom FileManager[FM]. Läsning sker när användaren vill logga in och aktiviteten undersöker då om information om tidigare lyckad inloggning finns sparad. Vid varje lyckad inloggning skriver LoginActivity[LA], återigen genom FileManager[FM], inloggningsnamn och en token till filen.

5.2.7. Resources
N/A

5.2.8. References
N/A

5.2.9. Processing
N/A

5.2.10. Data

Lagrar information om den senaste lyckade inloggningen, vilket innefattar användarnamn och token i textformat.

5.3 [FM]

FileManager.

5.3.1. Type

Klass.

5.3.2. Purpose

Att hantera alla filer och filoperationer i klienten.

Klassen är relevant för URD-krav där användarkonfiguration och andra bestående tillstånd behöver läsas eller skrivas, nämligen 3.1.3, 3.1.4 och 3.1.9.

5.3.3. Function

Innehåller ett antal metoder som utför filoperationer.

FileManager är en singleton.
`getInstance() : FileManager`

Metoder:
`writeFavorites(UserList) : void`
Skriver en lista med användare till fil.
`readUserList() : UserList`
Läser användarlistan från filen och skapar ett UserList[UL]-objekt.
`writeFavorites(UserList): void`
Skriver en lista med favoritanvändare till fil.
`readFavorites() : UserList`
Läser favoriter från fil.
`addFavorite(User) : void`
Lägger till favoriter till fil.
`removeFavorite(User) : void`
Tar bort favoriter från fil.
`writeLoginInfo(LoginInfo) : void`
Skriver inloggningsinformation till fil.
`readLoginInfo() : LoginInfo`
Läser inloggningsinformation från fil.
`readToken() : String`
Läser token från fil.
`writeToken(String) : void`
Skriver token till fil.
`setContext(Context) : void`
Behövs för att kunna skapa instanser av klassen.

5.3.4. Subordinates

FavoritesFile[FF] och LoginDetails[LD].

5.3.5. Dependencies
Applikationen behöver tillåtelse att läsa och skriva filer.

5.3.6. Interfaces
Klassen anropas för att lagra och skriva inloggningsinformation vilken inkluderar autentiseringstoken, inställningar och en lista med favoriter.

Andra komponenter som använder sig av FileManager:
FavoritesFragment[FFrag] anropar readFavorites() och readUserList() för att kunna presentera en lista med användarens favoriter. Nya favoriter läggs till med ett anrop till addFavorite(User[U]) och en favorit tas bort med ett anrop till removeFavorite(User[U]).

LoginActivity[LA] anropar metoderna som berör inloggningsinformation och token.

5.3.7. Resources
N/A

5.3.8. References
N/A

5.3.9. Processing
Klassen utnyttjar designmönstret *initialization-on-demand holder*, vilket är en trådsäker variant av singleton. Av praktiska skäl instansieras klassen som ett globalt objekt. Eftersom samtliga aktiviteter verkar oberoende av varandra, finns det inget uppenbart sätt att skicka objektet som referens.

Eftersom klassen saknar tillstånd är det en lämplig kandidat för singleton-mönstret. Kodexempel för detta designmönster presenteras i avsnittet om DBConn[DBC].

Filoperationer utförs med Javas standardbibliotek.

5.3.10. Data
Fält för filerna och sökvägar till filerna är:
```
favorites : File
logins : File
userlist : File
FAV_FILE_PATH : String
LOGIN_FILE_PATH : String
USERLIST_FILE_PATH : String
```

5.4 [BID]
BeaconID.

5.4.1. Type
Klass.

5.4.2. Purpose

Syftet är att tillhandahålla den mest relevanta informationen angående en beacon, exempelvis värdena major och minor. Objekt av denna klass ska kunna hämtas av ServerCon[SC] från BeaconWhisperer[BW] där objektet skapas.

Klassen är relevant för URD-kraven 3.1.1, 3.1.2, 3.1.5, 3.1.8 och 3.2.1.

5.4.3. Function
Klassen innehåller getters för de tre fälten.

Metoder:
```
getUuid() : String
getMajor() : int
getMinor() : int
getBatteryPercent() : int
```

Objekt av klassen skapas genom method chaining med hjälp av BeaconFactory[BF].

5.4.4. Subordinates
BeaconFactory[BF].

5.4.5. Dependencies
N/A

5.4.6. Interfaces
Klassen används som en behållare för beacon-specifik information, och värdena i objektet kan ändras och läsas av andra komponenter. Ett objekt av klassen hålls kontinuerligt uppdaterat i BeaconWhisperer[BW] som en representation av den närmaste beaconen, och kan vidarebefordras till ServerCon[SC] för att kontakta servern med relevant information.

5.4.7. Resources
N/A

5.4.8. References
Javadoc för det SDK som används för att kommunicera med beacons.

5.4.9. Processing
Getters. Exempel:

```
public int getMajor() {
	return major;
}
```

5.4.10. Data
Se Kapitel 1.3 för förklaring av fälten.
```
uuid : String
major : int
minor : int
batteryPercent : int
```

5.5 [BF]
BeaconFactory.

5.5.1. Type
Lokal klass.

5.5.2. Purpose
Att göra det smidigare att skapa BeaconID[BID]-objekt.

Klassen är relevant för samma URD-krav som för BeaconID[BID], nämligen 3.1.1, 3.1.2, 3.1.5, 3.1.8 och 3.2.1.

5.5.3. Function
Klassen innehåller setters för de tre fälten i BeaconID[BID].

Metoder:
```
setUuid(String) : void
setMajor(int) : void
setMinor(int) : void
setBatteryPercent(int) : BeaconFactory
```

Den lokala klassen skapar ett BeaconID[BID]-objekt via konstruktorn:
```
BeaconFactory() : BeaconFactory
```

5.5.4. Subordinates
N/A

5.5.5. Dependencies
N/A

5.5.6. Interfaces
Används av klasser som vill skapa BeaconID[BID]-objekt genom klassens konstruktor, nämligen BeaconWhisperer[BW].

5.5.7. Resources
N/A

5.5.8. References
Javadoc för det SDK som används för att kommunicera med beacons.

5.5.9. Processing
Setters. Exempel:

```
public void setMajor(int major) {
      this.major = major;
}
```

5.5.10. Data
`beacon : BeaconID`
BeaconID[BID]-objektet som skapas i konstruktorn.

5.6 [U]
User.

5.6.1. Type
Klass.

5.6.2. Purpose
Att representera en användare av applikationen som ett objekt med diverse fält. Klassen är relaterad till URD-kraven 3.1.4, 3.1.5, 3.1.6, 3.1.7, 3.1.10 och 3.1.11.

5.6.3. Function
Klassen innehåller getters som läser de privata fälten och motsvarande setters, samt en metod som kan returnera en JSON-representation av användaren.

Metoder:
`toJSONObject() : JSONObject`
Returnerar en JSON-representation av användaren.
```
getFirstName() : String
getLastName() : String
getUID() : String
getRoles() : List<String>
setFirstName(String) : void
setLastName(String) : void
setUID(String) : void
setRoles(List<String>) : void
toString(): String
```
Returnerar en sträng med användarens namn i ett specificerat format.

Ett objekt av klassen skapas antingen genom att anropa konstruktorn med förnamn, efternamn, användarnamn och en lista av roller, eller att anropa konstruktorn med ett JSON-objekt.
```
User(String, String, String, List<String>) : User
User(JSONObject) : User
```

5.6.4. Subordinates
N/A

5.6.5. Dependencies
Användaren måste finnas i databasen.

5.6.6. Interfaces
Klassen används som en lagringsbehållare för information om en användare. Ett objekt av klassen kan skapas och skickas mellan applikationens aktiviteter. Inflödet av data är när setters används, och i detta fall kommer ArrayLists och strängar att skickas som

parametrar. Utflödet är getters av samma typ, det vill säga ArrayLists och strängar som hämtas.

Ett exempel på klassens roll i applikationen är när SearchFragment[SF] anropar ServerCon[SC] som returnerar ett UserList[UL]-objekt. Denna UserList[UL] innehåller en lista av Users[U], och kan visas upp i någon vy.

5.6.7. Resources
N/A

5.6.8. References
N/A

5.6.9. Processing
Getters och setters. Exempel:

```
public List<String> getRoles() {
      return roles;
}
```

5.6.10. Data
Information om användaren:

```
firstName : String
lastName : String
uid : String
roles : List<String>
```

Information om JSON-objektet:

```
FIRST_NAME : String
LAST_NAME : String
UID : String
ROLES : String
```

5.7 [UL]
UserList.

5.7.1. Type
Klass.

5.7.2. Purpose
Finns för att det ska gå att skapa ArrayList<User>-objekt från ett JSON-objekt genom dess konstruktor.

Klassen är relevant för URD-kraven 3.1.4, 3.1.10, 3.1.11.

5.7.3. Function
Klassen representerar en användarlista, det vill säga en lista med User[U]-objekt. Klassen kan ta en JSONArray i dess konstruktor och kan därefter göra om denna till User[U]-objekt. Klassen möjliggör också konvertering från en lista med User[U]-objekt till en JSONArray.

ArrayList är en superklass till denna klass och därmed ärver klassen all funktionalitet från ArrayList.

Metoder:
`toJSONArray() : JSONArray`

Ett objekt av klassen kan skapas genom klassens konstruktor.
`UserList(JSONArray) : UserList`

5.7.4. Subordinates
User[U].

5.7.5. Dependencies
Klassen är endast beroende av JSONArray-objektet som konstruktorn tar som parameter.

5.7.6. Interfaces
ServerCon[SC] skapar ett UserList[UL]-objekt med ett JSONArray-objekt som parameter till klassens konstruktor. UserList[UL]-objektet blir då en ArrayList av Users[U]. Detta objekt används sedan för att kunna extrahera User[U]-objekt som kan användas av bland annat SearchFragment[SA] och FavoritesFragment[FA].

5.7.7. Resources
JSON.simple, ett bibliotek med öppen källkod för JSON encoding/decoding i Java.

5.7.8. References
Javadoc för JSONArray. [5]

5.7.9. Processing
Klassen kallar på Users[U]-konstruktor då den ska skapa User[U]-objekt från en JSONArray.

5.7.10. Data
`jsonArray : JSONArray`
Ett fält som innehåller det JSONArray-objekt som skickades in i konstruktorn.

5.8 [LE]
LogEntry.

5.8.1. Type
Klass.

5.8.2. Purpose
Klassen används för att representera ett inlägg i en användares positionsförfrågningslogg. Den är således direkt relevant för URD-kravet 3.1.7.

5.8.3. Function

Getters för ett Date och ett User[U]-objekt, vilka representerar en tidsstämpel respektive användaren som begärde positionen. Dessutom finns en toString-metod som skriver ut vem som sökt användaren vid vilken tid.

Metoder:

```
getTimestamp() : Timestamp
getRequester() : User
toString() : String
toJSONObject() : JSONObject
```

Konstruktorn skapar ett objekt innehållande tidpunkt för en positionsförfrågning och vem som gjort denna. Det finns även en konstruktor som utgår ifrån ett JSON-objekt.

```
LogEntry(Timestamp, User) : LogEntry
LogEntry(JSONObject) : LogEntry
```

5.8.4. Subordinates

N/A

5.8.5. Dependencies

Klassens syfte är att representera inlägg i positionsförfrågningsloggen, vilken finns lagrad i en extern databas. Databasen måste alltså finnas.

5.8.6. Interfaces

Allt dataflöde till objekt av klassen sker via getters och setters. Används av andra klasser i form av fragmentet LogBookFragment[LBF] via LogEntryList[LEL]. LogEntryList[LEL] skapar en lista med dessa objekt, vilken kan ses som behållare för inlägg i positionsförfrågningsloggen.

5.8.7. Resources

N/A

5.8.8. References

N/A

5.8.9. Processing

Getters. Exempel:

```
public Date getTimestamp() {
	return timestamp;
}

public User getRequester() {
	return requester;
}
```

5.8.10. Data

```
timestamp : Timestamp
```

Lagrar tiden för positionsförfrågningen.

```
requester : User
```

Lagrar användaren som gjorde positionsförfrågningen.

5.9 [LEL]
LogEntryList.

5.9.1. Type
Klass.

5.9.2. Purpose
Finns för att det ska gå att skapa ArrayList<LogEntry> från ett JSON-objekt genom dess konstruktor. Gör det smidigare att hantera listor av LogEntry[LE] och är därför relevant för samma URD-krav som den klassen, det vill säga 3.1.7.

5.9.3. Function
Är en subklass av ArrayList och ärver därför dess metoder, analogt med UserList[UL]. Konstruktorn tar en JSONArray och gör om denna till LogEntry[LE]-objekt. Klassen möjliggör också konvertering från en lista med LogEntry[LE]-objekt till en JSONArray.

Metoder:
`toJSONArray() : JSONArray`
Konverterar en LogEntryList[LEL] till en JSON-representation.

Objekt av klassen skapas genom följande konstruktor.
`LogEntryList(JSONArray) : LogEntryList`

5.9.4. Subordinates
Klassen är en lista av LogEntrys[LE], vilket då är en underlydande klass.

5.9.5. Dependencies
Klassen är endast beroende av JSONArray-objektet som konstruktorn tar som parameter.

5.9.6. Interfaces
ServerCon[SC] skapar ett LogEntryList-objekt med ett JSONArray-objekt som parameter till klassens konstruktor. LogEntryList-objektet blir då en ArrayList av LogEntry[LE]. Detta objekt används sedan för att kunna extrahera LogEntry[LE]-objekt, som i sin tur används av LogbookFragment[LBF].

5.9.7. Resources
JSON.simple.

5.9.8. References
Javadoc för JSONArray. [5]

5.9.9. Processing
Klassen kallar på LogEntrys[LE] konstruktor när den ska skapa LogEntry[LE]-objekt utifrån en JSONArray.

5.9.10. Data

`jsonArray : JSONArray`
Ett fält som innehåller det JSONArray-objekt som skickades in i konstruktorn.

5.10 [LINFO]

LoginInfo.

5.10.1 Type

Klass.

5.10.2. Purpose

Klassen innehåller all information som behövs för att kunna logga in i applikationen.

Klassen är relaterad till URD-krav 3.1.6 eftersom man måste veta vem som gör förfrågan för att kunna logga den.

5.10.3. Function

Getters och setters för de privata fälten.

Metoder:

```
getUserName() : String
getPass() : String
getToken() : String
getRememberMe() : boolean
setUserName(String) : void
setPass(String) : void
setToken(String) : void
setRememberMe(boolean) : void
toJSONObject : JSONObject
```

Konstruktorn som skapar ett objekt av klassen tar tre strängar, som motsvarar användarnamn, lösenord respektive token. Det finns även en konstruktor som utgår ifrån ett JSONObjekt.

```
LoginInfo(String, String, String) : LoginInfo
LoginInfo(JSONObject) : LoginInfo
```

5.10.4. Subordinates

N/A

5.10.5. Dependencies

Ett användarnamn motsvarande det i klassen bör finnas i databasen, och analogt för lösenordet.

5.10.6. Interfaces

Allt dataflöde till objekt av klassen, det vill säga hämtning och modifikation av klassens fält, sker via getters och setters. Ett objekt av klassen skapas i LoginActivity[LA] och vidarebefordras till ServerCon[SC] som i sin tur kontaktar servern och försöker logga in.

5.10.7. Resources

N/A

5.10.8. References
N/A

5.10.9. Processing
Getters. Exempel:

```
public String getPass() {
      return pass;
}
```

5.10.10. Data
```
pass : String
userName : String
token : String
rememberMe : boolean
```

5.11 [BW]
BeaconWhisperer.

5.11.1. Type
Klass.

5.11.2. Purpose
Syftet är att hålla koll på informationen om närmaste beacon och hålla denna uppdaterad.

Klassen är relevant för URD-kraven 3.1.1, 3.1.5, 3.1.8, 3.2.1, 3.2.8 och 3.2.9.

5.11.3. Function
Startar internt en bakgrundsprocess som skannar efter beacons och uppdaterar ett fält som innehåller information om den närmaste påträffade beaconen. Denna bakgrundsprocess körs så länge användaren är inloggad. ServerCon[SC] ska kunna hämta information angående närmaste beacon. Innehåller metoder som genom SDK:t kan läsa av information om beacons i dess närhet.

<u>Metoder:</u>
`startRanging() : void`
Börjar söka efter beacons i bakgrunden och uppdaterar konsekvent närmaste beacon.
`stopRanging() : void`
Slutar söka efter beacons i bakgrunden.
`promptBlueTooth(MainActivity) : void`
Startar en dialog med användaren (med hjälp av MainActivity[MA]-objektet) om Bluetooth inte är igång och ber i sådana fall användaren att slå på Bluetooth.
`onActivityResult(MainActivity, int, int, Intent) : void`
Hanterar resultatet av Bluetooth-dialogen, det vill säga vad som ska hända om användaren tillät applikationen att starta Bluetooth eller inte.

`getClosestBeacon() : BeaconID`

Objekt av klassen skapas med följande parameterlösa konstruktor:
`BeaconWhisperer() : BeaconWhisperer`

5.11.4. Subordinates
BeaconID[BID].

5.11.5. Dependencies
N/A

5.11.6. Interfaces
MainActivity[MA] kommer instansiera klassen och använda den för att be användaren slå på Bluetooth om det inte redan är på. Genom det SDK som hör till de beacons som används hämtas information om närmaste beacon. Denna information sparas i ett BeaconID[BID]-objekt, vilket ServerCon[SC] sedan kan använda. ServerCon[SC] kommer alltså främst använda metoden getClosestBeacon().

5.11.7. Resources
SDK för den typ av beacon som används.

5.11.8. References
Javadoc för det SDK som används för att kommunicera med beacons.

5.11.9. Processing
Använder BeaconID:s[BID] getters för att få information om beacons. Klassen skickar ett BeaconID[BID]-objekt som motsvarar närmaste beacon till ServerCon[SC] vid begäran.

5.11.10. Data
`bid : BeaconID`
Lagrar relevant information om närmaste beacon.
`bm : BeaconManager`
Används för att kommunicera med beacons.
`REQUEST_ENABLE_BT : int`
En kod som används av Androidsystemet för att be användaren slå på Bluetooth om det inte redan är på.

5.12 [SC]
ServerCon.

5.12.1 Type
Klass.

5.12.2 Purpose
Behövs för URD-krav där klienten skall interagera med servern, nämligen 3.1.4, 3.1.5, 3.1.6 och 3.1.7.

5.12.3 Function
Hantera all kommunikation med servern, som exempelvis inloggning samt begäran av position och logg.

Klassen innehåller den lokala klassen ConnectionHandle[CH]. Denna har en konstruktor `ConnectionHandle(String, int, String) : ConnectionHandle`, där parametrarna motsvarar användarnamn, lösenord respektive token.

Metoder:
`sendPosition(int, int) : boolean`
Skickar den inloggade användarens position.
`requestPosition(User) : String`
`requestPosition(User, int) : String`
Frågar servern om en användares position.
`requestLog() : LogEntryList`
Frågar servern om en användares positionsförfrågningslogg.
`login(LoginInfo, String) : boolean`
Försöker logga in en användare.
`sendBatteryPercent(BeaconID) : boolean`
Skickar batteriprocenten för en beacon.
`getSocket(int) : Socket`
Hämtar den socket som kan användas för att skicka och ta emot data från servern.
`logout() : boolean`
Loggar ut från servern så att ingen kommunikation kan fortgå.
`getFavorites() : UserList`
Hämtar alla favoritanvändare för den inloggade användaren.
`getUsers(String) : UserList`
Hämtar alla de användarna som är online och uppfyller ett givet mönster, som exempelvis namn som börjar på “A”.
`getConnection() : ConnectionHandle`
Ger ett handle för att kunna kommunicera med servern.
`getErrorMessage() : String`
`flush() : void`
Tar bort nuvarande anslutning till servern.
`isInvalidated() : boolean`
Undersöker om nuvarande anslutning är giltig.
`ping() : boolean`
Pingar servern och talar om ifall servern är tillgänglig eller ej.

5.12.4 Subordinates
LoginInfo[LINFO] och ConnectionHandle[CH].

5.12.5 Dependecies
BeaconWhisperer[BW] och FileManager[FM].

5.12.6 Interfaces
Innehåller metoder både för att begära information från servern och skicka information till den. ClientOutputFormatter[COF] används för att formatera informationen korrekt. Metoden

getConnection() används för att etablera kontakt med servern som tillåter denna kommunikation.

Data som skickas från server till klient kommer att skickas som JSON-strängar. Dessa data kommer att konverteras tillbaka till rätt typ av objekt och returneras till begärande Activity eller Fragment av ServerCon[SC]. Vid inloggning kommer servern exempelvis att skicka en lista med inloggade användare, vilka kommer att plockas ut från databasen på servern, omvandlas till JSON och skickas genom socketen. Klienten tar emot dessa data, och skapar ett UserList[UL]-objekt som sedan kan användas i klienten.

Situationen då klienten vill skicka till informationen till servern är analog.

5.12.7 Resources
Behöver en server att ansluta till för informationsutbyte.

5.12.8 References
N/A

5.12.9 Processing
Denna komponent skapas vid inloggning när användaren har matat in användarnamn och lösenord i form av en ConnectionHandle[CH] som anslutning till server.

```
serverCon.login(LoginInfo);
```

Om inloggningen misslyckas måste man försöka igen. Varje objekt innehåller användarnamn och lösenord, och ServerCon[SC] håller själv reda på tokens. Metoden login måste anropas innan data kan hämtas från servern. Om anslutningen misslyckas, eller login inte har anropats innan data begärs från servern, returneras null. Annars kan data nu skickas och tas emot från servern med hjälp av metoder.

5.12.10 Data
`handle : ConnectionHandle`
Ett fält där objekt av den lokala klassen kan sparas.
`errorMessage : String`
`invalidated : boolean`
Ett fält som talar om statusen på anslutningen, om den är valid eller inte.
`IP : String`
Lagrar IP-adressen till servern.
`PORT : int`
Lagrar porten till servern.
`TIMEOUT : int`
Standard timeout-tiden för anslutningen.
`TAG : String`
Används för att identifiera loggmeddelanden från den här klassen i logcat.

5.13 [CH]
ConnectionHandle.

5.13.1. Type
Lokal klass.

5.13.2. Purpose
Används för att skapa en anslutning till servern.

Är en utvigdning till ServerCon[SC] och är således relevant för samma krav som den klassen, nämligen 3.1.4, 3.1.5, 3.1.6 och 3.1.7.

5.13.3. Function
Klassen innehåller inga metoder utan används istället som en behållare för en anslutning med tre fält för användarnamn, lösenord och en token.

Konstruktorn tar ett användarnamn, ett lösenord och en token.
```
ConnectionHandle(String, int, String) : ConnectionHandle
```

5.13.4. Subordinates
N/A

5.13.5. Dependencies
N/A

5.13.6. Interfaces
Objekt av klassen används vid anslutning till servern.

5.13.7. Resources
N/A

5.13.8. References
N/A

5.13.9. Processing
N/A

5.13.10. Data
```
token : String
uid : String
password : int
```

5.14 [COF]
ClientOutputFormatter.

5.14.1. Type
Klass.

5.14.2. Purpose

Att se till så att data som skickas från klient till server är ordentligt formatterad.

Klassen är relevant för URD-kraven 3.1.4, 3.1.5, 3.1.6 och 3.1.7.

5.14.3. Function
Bygger ett JSONObject-objekt som kan göras om till en JSON-sträng vilken kan tolkas korrekt av servern. Använder ett antal metoder för att bygga detta objekt med method chaining.

Metoder:
`addData(String, String) : ClientOutputFormatter`
Lägger till ett nyckel/värde-par till JSONObject-objektet där värdet är en sträng.
`addJsonArray(String, JSONArray) : ClientOutputFormatter`
Lägger till ett nyckel/värde-par till JSONObject-objektet där värdet är en JSONArray.
`addJsonObject(String, JSONObject) : ClientOutputFormatter`
Lägger till ett nyckel/värde-par till JSONObject-objektet där värdet är ett JSONObject.
`mergeJsonObject(JSONObject) : ClientOutputFormatter`
Lägger till samliga nyckel/värde-par från ett annat JSONObject i JSONObject-objektet.
`toString() : String`

Konstruktorn initialiserar ett JSONObject-objekt:
`ClientOutputFormatter() : ClientOutputFormatter`

5.14.4. Subordinates
N/A

5.14.5. Dependencies
N/A

5.14.6. Interfaces
Ett objekt av klassen skapas i ServerCon[SC] som bygger en JSON-sträng vilken sedan skickas till servern.

5.14.7. Resources
JSON.simple.

5.14.8. References
Javadoc för JSON.simple. [5]

5.14.9. Processing
Anropar metoder från det externa JSON.simple-paketet för att bygga JSON-strängar.

5.14.10. Data
`json : JSONObject`

5.15 [LA]
LoginActivity.

5.15.1. Type
Klass.

5.15.2. Purpose
Klassen utvidgar Activity och ska kunna visa en inloggningsvy där det är möjligt för användaren att logga in. Inloggningen kan även ske automatiskt om användaren tidigare angivit korrekta uppgifter. Klassen ska kunna kommunicera med ServerCon[SC] för att verifiera huruvida de angivna inloggningsuppgifterna är korrekta eller ej.
Klassen är relevant för URD-kraven 3.1.3, 3.2.2 och 3.2.7.

5.15.3. Function
Klassen innehåller funktioner som hanterar inloggning, och behöver därför kunna kommunicera med databasen via ServerCon[SC]. Klassens huvudmetod login anropar metoden validate, som kan tala om huruvida inloggningen lyckades eller ej. Klassen innehåller metoder för det grafiska gränssnittet och för att hämta de senaste lyckade inloggningsuppgiterna genom FileManager[FM].

Metoder:
`login() : void`
`validate() : void`
Kolla om inloggningsinformationen är giltig.
`updateLoginInfo() : void`
Uppdatera inloggningsinformationen utifrån de uppgifter som användaren angivit.
`onCreate(Bundle) : void`
`onCreateOptionsMenu(Menu) : boolean`
`onButtonClicked(View) : void`
Definiera vad som ska hända när användaren klickar på inloggningsknappen.
`onCheckboxClicked(View) : void`
Definiera vad som ska hända när användaren klickar på “Kom ihåg mig”-kryssrutan.

5.15.4. Subordinates
LoginInfo[LINFO] och LoginDetails[LD].

5.15.5. Dependencies
MySQL-databas vars innehåll används för att verifiera inloggningar, och filen LoginDetails[LD].

Eftersom denna klass är en Activity med GUI-element kommer det krävas diverse XML-filer som är kutym inom Androidprogrammering, som exempelvis en XML-fil innehållande aktivitetens layout.

5.15.6. Interfaces
Klassen visar den grafiska inloggningsvyn vilken innehåller två textfält, en inloggningsknapp och en "Kom ihåg mig"-kryssruta. När användaren valt sina inloggningsuppgifter och tryckt på inloggningsknappen skickas informationen till klassen ServerCon[SC]. Tillbaka fås en token som indikerar huruvida inloggningsuppgifterna var korrekta eller ej. Om uppgifterna var korrekta loggas användaren in, annars ges en ny möjlighet att ange inloggningsuppgifter.

5.15.7. Resources
N/A

5.15.8. References
N/A

5.15.9. Processing
Pseudokod:
```
login()
    validate()
    IF exists information in LoginDetails THEN
        Login
    ELSE
        Input userinformation
        Server checks if valid
        IF valid THEN
                    Login and proceed to MainActivity[MA]
        ELSE
                    Stay at startscreen
```

5.15.10. Data
`linfo : LoginInfo`
Fältet lagrar inloggningsinformation som kan användas till inloggning.
`u : EditText`
Fältet motsvarar det textinskrivningsfält där användaren skriver sitt användarnamn.
`p : EditText`
Fältet motsvarar det textinskrivningsfält där användaren skriver sitt lösenord.
`cb : CheckBox`
Fältet motsvarar lådan där användaren kan markera om applikationen ska komma ihåg inloggningsuppgifter.

5.16 [MA]
MainActivity.

5.16.1. Type
Klass.

5.16.2. Purpose
När användaren har lyckats logga in krävs det att denne hamnar någonstans, nämligen i denna klass. Syftet är alltså att vara standardvyn för inloggade användare.

Är ett bra sätt att grafiskt presentera de olika Fragments, nämligen FavoritesFragment[FFrag], SearchFragment[SF] och LogbookFragment[LBF], och är således indirekt relevant för deras URD-krav. Utöver det är klassen även direkt relevant för URD-kraven 3.1.3 i kombination med 3.2.7, det vill säga ett distraktionsfritt och smidigt användargränssnitt.

5.16.3. Function

Innehåller tre Fragments som flikar och behöver därför metoder som stöder detta grafiskt. Kan även starta SettingsActivity[SA]. Klassen utvigdar Activity och ärver således dess funktionalitet.

Metoder:

```
onCreate() : void
onCreateptionsMenu(Menu) : boolean
onPageSelected(int) : void
onPageScrolled(int, float, int) : void
onPageScrollStateChanged(int) : void
onBackPressed() : void
onTabReselected(Tab, FragmentTransaction) : void
onTabSelected(Tab, FragmentTransaction) : void
onTabUnselected(Tab, FragmentTransaction) : void
onOptionsItemSelected(MenuItem) : boolean
openSettings() : void
```

Öppnar inställningsvyn.

```
checkIfLogOut() : void
```

Öppnat en dialogruta som frågar om användaren är säker på att denne vill logga ut.

```
getContext() : Context
```

Används av klasser som behöver hämta Context utan att vara en Activity eller Fragment.

5.16.4. Subordinates

BeaconWhisperer[BW], FavoritesFragment[FFrag], SearchFragment[SF], LogbookFragment[LBF] och SettingsActivity[SA].

5.16.5. Dependencies

Behöver vara inloggad och därmed startas från LoginActivity[LA], och alltså måste LoginActivity[LA] finnas. Klassen presenterar information som tagits fram av andra klasser, och är därför bara indirekt beroende av dem.

5.16.6. Interfaces

Klassen är det centrala navet i applikationen eftersom användaren kommer dit efter inloggning. Härifrån kan användaren ta sig till alla de andra vyerna och tillbaka. Klassens logik sker i de flikar som finns i form av FavoritesFragment[FFrag], SearchFragment[SF] och LogBookFragment[LBF].

5.16.7. Resources

N/A

5.16.8. References

N/A

5.16.9. Processing

Klassen väljer vilken av de tre valbara Fragments som finns som ska presenteras. Det är dessutom möjligt att gå till SettingsActivity[SA].

5.16.10. Data

```
viewPager : ViewPager
```

Visar XML-element.

`tabsPagerAdapter : TabsPagerAdapter`
Adapter som används för att hantera fragments.
`actionBar : ActionBar`
`mContext : Context`
`bw : BeaconWhisperer`
Används för att hantera beacons.

5.17 [FFrag]

FavoritesFragment.

5.17.1. Type

Klass.

5.17.2. Purpose

Klassens syfte är att snabbt kunna hitta de personer som användaren ofta söker efter. Klassen är således relevant för URD-kraven 3.1.3, 3.1.12, 3.2.4 och 3.2.7.

5.17.3. Function

Klassen utvigdar ListFragment och ärver således dess funktionalitet. Klassen overridar metoder som möjliggör en grafisk vy på användarens favoriter. Det ska från denna vy gå att nå favoriternas profilsidor och det finns således metoder som startar en PersonActivity[PA].

Metoder:
`onActivityCreated(Bundle) : void`
`onCreateView(LayoutInflater, ViewGroup, Bundle) : boolean`
`onListItemClick(ListView, View, int, long) : void`

5.17.4. Subordinates

User[U].

5.17.5. Dependencies

En användare måste vara inloggad för att kunna hämta sina favoriter. MainActivity[MA] måste finnas för att klassen ska kunna vara en del av denna.

Eftersom denna klass är ett Fragment med GUI-element kommer det krävas diverse XML-filer som är kutym inom Androidprogrammering, som exempelvis en XML-fil innehållande layouten.

5.17.6. Interfaces

En lista med favoritanvändare, i form av User[U]-objekt, hämtas från ServerCon[SC]. Innehållet i dessa User[U]-objekt visas upp i en lista. När användaren klickar på ett User[U]-objekt öppnas den tillhörande PersonActivityn[PA].

5.17.7. Resources

N/A

5.17.8. References

N/A

5.17.9. Processing
Använder sig av User[U]-objektens getters för att kunna visa upp relevant data.

5.17.10. Data
`users : List<User>`
Ett fält som innehåller en lista med favoritanvändarna.

5.18 [SF]
SearchFragment.

5.18.1. Type
Klass.

5.18.2. Purpose
Motsvarar ett av applikationens huvudsyfte, nämligen att kunna söka efter och filtrera andra användare och få ut en lista med dessa på skärmen.

Klassen blir därför relevant för många av de URD-krav som finns, nämligen 3.1.3, 3.1.4, 3.1.10, 3.1.11, 3.1.12 och 3.2.4.

5.18.3. Function
Har metoder som filtrerar listan av alla användare efter vad den sökande användaren har skrivit in i sökfältet och vilka av rollerna som är markerade för tillfället. Metoden displayResults visar upp denna filtrerade lista på skärmen. Klassen utvigdar Fragment och ärver således dess funktionalitet.

Metoder:
```
onActivityCreated(Bundle) : void
onCreateView(LayoutInflater, ViewGroup, Bundle) : boolean
onListItemClick(ListView, View, int, long) : void
```
Definierar vad som ska hända när användaren klickar på ett sökresultat.
```
filterByName(): UserList
filterByRoles() : UserList
displayResults() :void
```

5.18.4. Subordinates
Använder sig av User[U]-objekt som element i den filtrerade listan som klassen ska visa upp.

5.18.5. Dependencies
Klassen kommunicerar med ServerCon[SC] för att komma åt listan av användare. MainActivity[MA] måste finnas för att SearchFragment[SF] ska kunna vara en tabb till denna.

Eftersom denna klass är ett Fragment med GUI-element kommer det krävas diverse XML-filer som är kutym inom Androidprogrammering, som exempelvis en XML-fil innehållande layouten.

5.18.6. Interfaces
Hämtar data ifrån User[U]-objekt med hjälp av getters. När klassen instansieras från MainActivity[MA] så skickas en begäran genom ServerCon[SC] till servern om en lista på samtliga användare. När denna lista har hämtats sparas den i klassen.

5.18.7. Resources
N/A

5.18.8. References
N/A

5.18.9. Processing
Klassen visar upp en lista av User[U]-objekt på skärmen och hämtar data ifrån dessa med getters.

5.18.10. Data
`roles : ArrayList<String>`
En lista över de roller användaren har markerat.
`name : String`
Vad som står i namn-sökfältet.
`original : List<User>`
Den ofiltrerade listan av alla användare.
`filteredRoles : List<User>`
En delmängd av fältet original, filtrerad efter de roller användaren har markerat.
`filteredName : List<User>`
En delmängd av fältet filteredRoles, filtrerad efter vad det står i namn-sökfältet.

5.19 [LBF]
LogbookFragment.

5.19.1. Type
Klass.

5.19.2 Purpose
Är ett Fragment som visar en visuell representation av användarens positionsförfrågningslogg, det vill säga en lista över vilka andra användare som har begärt ens position och när, och som sträcker sig en bestämd tid bakåt i tiden.

Klassen är relevant för URD-kraven 3.1.3, 3.1.6, 3.1.7 och 3.1.12.

5.19.3. Function
Har en metod fetchLogBook som anropar ServerCon[SC] som i sin tur returnerar en LogEntryList[LEL], vilket är användarens logg från databasen. Har också en metod displayLogBook som visar upp den hämtade loggen på skärmen. Utvigdar Fragment och ärver således dess funktionalitet.

<u>Metoder:</u>
`onCreateView(LayoutInflater) : View`

`onActivityCreated(Bundle) : void`
`fetchLogBook() : LogEntryList`
Hämta loggboken från servern.
`displayLogBook() : void`
`onListItemClicked() : void`
Definiera vad som ska hända när användaren klickar på ett listelement.

5.19.4. Subordinates
Använder sig av LogEntry[LE] för att representera inlägg i förfrågningsloggen.

5.19.5. Dependencies
Loggen ligger lagrad i databasen och hämtas ut via klassen ServerCon[SC].
MainActivity[MA] måste finnas för att klassen ska kunna vara en del av denna.

Eftersom denna klass är ett Fragment med GUI-element kommer det krävas diverse XML-filer som är kutym inom Androidprogrammering, som exempelvis en XML-fil innehållande layouten.

5.19.6. Interfaces
Är ett Fragment i applikationen som hör till MainActivity[MA] och som användaren kan interagera med. Kommunicerar med servern via ServerCon[SC] för att kunna hämta loggar.

5.19.7. Resources
Databasen som ServerCon[SC] hämtar sin information ifrån.

5.19.8. References
N/A

5.19.9. Processing
Informationen som finns lagrad i en LogEntryList[LEL] hämtas i form av LogEntrys[LE] och visas upp med hjälp av getters.

5.19.10. Data
N/A

5.20 [SA]
SettingsActivity.

5.20.1. Type
Klass.

5.20.2. Purpose
Behövs för att användaren ska kunna skräddarsy applikationen efter individuell preferens.

Klassen är relevant för URD-kraven 3.1.3 och 3.1.9.

5.20.3. Function

Kryssrutor för att kunna skräddarsy appen. Klassen utvigdar Activity och ärver således dess funktionalitet.

Metoder:
`onCreate(Bundle) : void`

5.20.4. Subordinates
N/A

5.20.5. Dependencies
I och med att funktionaliteten utökas kan det komma att behövas en lagringsplats för inställningar. Settings sparas via SharedPreferences [7] i Android.

Eftersom denna klass är en aktivitet med GUI-element kommer det krävas diverse XML-filer som är kutym inom Androidprogrammering, som exempelvis en XML-fil innehållande layouten.

5.20.6. Interfaces
Aktiviteten startas från MainActivity[MA]. Aktiviteten möjliggör ändringar i hela applikationen.

5.20.7. Resources
N/A

5.20.8. References
Javadoc för SharedPreferences. [7]

5.20.9. Processing
N/A

5.20.10. Data
Lagrar den data som behövs beroende på vilka inställningar som kommer att implementeras.

5.21 [PA]
PersonActivity.

5.21.1. Type
Klass.

5.21.2. Purpose
Samtliga användare behöver ha en profilsida där andra kan gå in och få information om användarens roll och nuvarande position, samt eventuellt lägga till/ta bort denna som favoritanvändare. Denna klass är till för att grafiskt visa dessa användarprofiler.

Klassen är relevant för URD-kraven 3.1.3, 3.1.5 och 3.2.7.

5.21.3. Function

Klassen behöver kunna kommunicera med ServerCon[SC] för att få information om var personen som söks befinner sig, och därför behövs metoden locateUser(). Den behöver också kunna åskådliggöra användarnas profilsidor och innehåller därför bland annat metoden displayUser() som gör detta. Det ska också gå att lägga till en användare som favorit, och därför finns metoder där kommunikation med FileManager[FM] och databasen via ServerCon[SC] äger rum. Klassen utvidgar Activity och ärver således funktionalitet från den.

Metoder:

```
locateUser() : void
displayUser() : void
listToString(ArrayList<String>, String) : String
```

Skapar en sträng från en ArrayList av roller.

```
toggleButton(View) : void
```

Gör lämpliga åtgärder när "Lägg till favorit"-knappen används.

```
isFavorite() : boolean
```

Avgör huruvida användaren vars profil som besöks är en favorit eller ej.

```
onCreate(Bundle) : void
onCreateOptionsMenu(Menu) : boolean
callUser(View view) : void
```

Gör lämpliga åtgärder för att ringa användaren.

5.21.4. Subordinates

User[U].

5.21.5. Dependencies

MySQL-databas vars innehåll används för att få positionsinformation. Dessutom måste databasen finnas för att tillåta en uppdatering av favoritanvändare från klassen.

Andra beroenden är:

- Beacons som kan ge positionsdata.
- Personen som söks behöver vara ansluten till systemet.
- En server som kan hantera positionsbegäran.

Eftersom denna klass är en Activity med GUI-element kommer det krävas diverse XML-filer som är kutym inom Androidprogrammering, som exempelvis en XML-fil innehållande layouten.

5.21.6. Interfaces

När ett objekt av klassen skapas får objektet, genom ett intent, den användare som ska visas i form av ett User[U]-objekt vars profil ska visas. Detta User[U]-objekt omvandlas sedan till information som kan visas grafiskt.

För att få andra användares position kommunicerar klassen med ServerCon[SC]. ServerCon[SC] skickar positionsdata för det User[U]-objekt som valts till användaren som sökte. Information visas sedan i den söktas användarprofil tillsammans med dennes roll. ServerCon[SC] behövs även för att lägga till eller ta bort en användare ur favoriter. Dessutom måste en kommunkation med FileManager[FM] finnas för att nuvarande favoriter

ska kunna läsas och vid uppdatering även kunna skrivas till FavoritesFile[FF].

5.21.7. Resources
N/A

5.21.8. References
N/A

5.21.9. Processing
I och med att aktiviteten startas får den ett User[U]-objekt inkluderat i dess intent. Detta User[U]-objekts getters används för att extrahera data som sedan kan visas grafiskt. User[U]-objektet används även som parameter vid positionsbegäran.

5.21.10. Data
`user : User`
Lagrar användaren som åskådliggörs.
`location : String`
En textsträng med rumsnamn.
`nameView : TextView`
Användarprofilens namn skrivs här.
`roleView : TextView`
Användarprofilens roller skrivs här.
`posView : TextView`
Användarprofilens position skrivs här.
`imgView : ImageView`
Användarprofilens profilbild visas här.
`toggleFav: Button`
Knapp för att lägga till/ta bort användaren som favorit.
`searchButton : Button`
Knapp för att söka efter användarens position.
`favorite : boolean`
Används för att tala om ifall användarprofilen som besöks är en favorit eller ej.

5.22 [SOF]
ServerOutputFormatter.

5.22.1. Type
Klass.

5.22.2. Purpose
Formatterar data som ska skickas från server till klient.

Behövs för de URD-krav där servern ska svara på förfrågningar från klienten, nämligen 3.1.4, 3.1.5, 3.1.7 och 3.1.12.

5.22.3. Function
Skapar en JSON-sträng från indata som förses med metoder som tillhör klassen.

Metoder:
`addData(String, String) : ServerOutputFormatter`
Lägger till ett nyckel/värde-par där värdet är en sträng.
`addMap(String, Map<String, String>) : ServerOutputFormatter`
Lägger till ett nyckel/värde-par där värdet är en map.
`addMapList(String, List<Map<String, String>>) : ServerOutputFormatter`
Lägger till ett nyckel/värde-par där värdet är en lista.
`toString() : String`

Konstruktorn är parameterlös:
`ServerOutputFormatter() : ServerOutputFormatter`

5.22.4. Subordinates
N/A

5.22.5. Dependencies
N/A

5.22.6. Interfaces
Samtliga Procedures som behöver skicka data till en klient anropar först klassen för att formattera datat till ett enhetligt format innan dessa skickas.

Data läggs till efter behov av den Procedure[P] som anropar klassen med metoderna addData, addMap och addMapList för att konstruera en JSON-sträng.

5.22.7. Resources
JSON.simple.

5.22.8. References
Javadoc för JSON.simple. [5]

5.22.9. Processing
JSON-strängen byggs upp i insättningsordning. Varje metodanrop bygger på JSON-trädet med hjälp av JSON.simple-biblioteket.

5.22.10. Data
`json : JSONObject`
De data som ska skickas från servern till klienten.

5.23 [L]
Listener.

5.23.1 Type
Klass.

5.23.2 Purpose

Behövs för de URD-krav där klienten ska interagera med servern, nämligen 3.1.4, 3.1.5, 3.1.7 och 3.1.12.

5.23.3 Function
Klassen är en serverdemon ansvarig för att allokera resurser som behövs för att starta servern, lyssna på en port och hantera inkommande anslutningar.

Klassen skapas när servern startas, vilket görs genom följande kommando:
```
> java Listener
```
En parameter för debug kan användas, nämligen `-d`.

Metoder:
```
main(String[]) : void
startServer() : void
```

5.23.4 Subordinates
AddFavoriteProcedure[AFP], DeleteFavoriteProcedure[DFP], LoginProcedure[LoginP], PositionResponseProcedure[PResP], PositionRequestProcedure[PReqP], SearchProcedure[SP], DBConn[DBC] och ListenerThread[LT].

5.23.5 Dependecies
N/A

5.23.6 Interfaces
Lyssnar på en port till vilken klienten kan ansluta till, och all kommunikation med servern ska ske via denna port. Servern håller inte några permanenta anslutningar; klienten måste skapa en ny socket varje gång servern ska kontaktas och se till att stänga socketen när kommunikationen är klar. Klienten börjar att skicka data. Data skickas över HTTPS med hjälp av JSON (se Kapitel 5.23.9 för möjliga värden som kan skickas).

Klassen är ansvarig för att skapa nya trådar (vilket sker med den lokala privata klassen ListenerThread[LT]), skräpinsamla gamla anslutningar som inte längre används och instansiera rätt Procedure[P]-klass med rätt parametervärden. Klassen skickar aldrig data tillbaka till klienten; detta görs i respektive Procedure[P]-klass.

5.23.7 Resources
Kräver serverhårdvara, JRE, internetanslutning samt port 31415 öppen och ledig.

5.23.8 References
N/A

5.23.9 Processing
Lyssnar på portnummer 31415.

Vid inkommande anslutning, acceptera anslutningen och skapa en pekare till den socket på vilken klienten kan skicka data:

```
Socket clientSocket = serverSocket.accept();
```

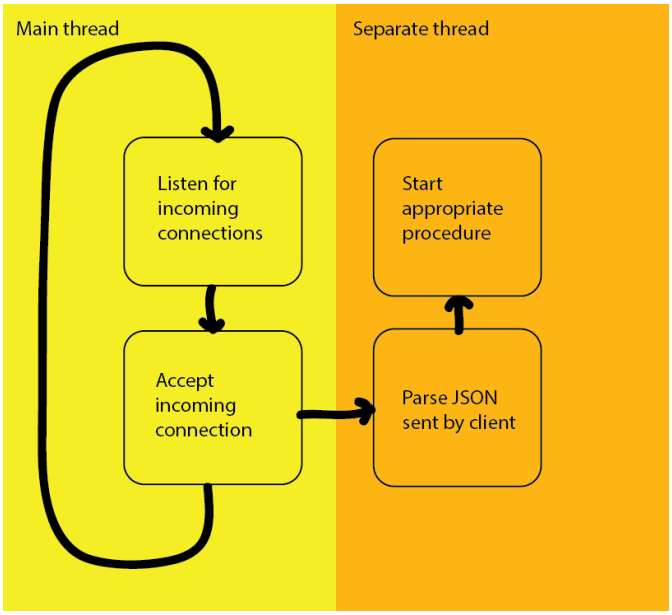
Klienten ska nu skicka data med JSON. Klassen parsar dessa data, och fältet action ska innehålla namnet på den procedur som skall köras på servern.

Möjliga värden på action: REQUEST_LOGIN, REQUEST_LOGOUT, REQUEST_POSITION, POSITION_RESPONSE, REQUEST_LOG, REQUEST_SEARCH_RESULTS, REQUEST_FAVORITES, REQUEST_FAVORITES_REMOVAL.

Som ett exempel antar vi att klienten vill logga in. Då skickas följande JSON-objekt:

```
{
   ...
   "action":"REQUEST_LOGIN"
   ...
}
```

Klientanslutningen hanteras internt i en separat tråd, ListenerThread[LT], vilket illustreras i figuren nedan. I steg 3 (Parse JSON sent by client) läses fältet action och kontroll görs av token med hjälp av DBConn[DBC]. Servern läser detta värde och startar rätt procedur, vilket i detta exempel är en LoginProcedure[LoginP]. För alla förfrågningar som inte är en inloggning, måste lyssnaren själv kontrollera att rätt token har angetts av klienten.



5.23.10 Data

```
debug : int
PORT : int
serverSocket : ServerSocket
pendingRequests : HashMap<String, Socket>
```

5.24 [LT]
ListenerThread.

5.24.1. Type
Lokal klass.

5.24.2. Purpose
Tråd som lyssnar och kommunicerar med servern så att flera användare kan kommunicera med servern samtidigt.

Används av Listener[L] och är således relevant för samma URD-krav som den, nämligen 3.1.4, 3.1.5, 3.1.7 och 3.1.12.

5.24.3. Function
Skapar en tråd som körs med metoden run.

Metoder:
`run() : void`
Kör tråden och gör lämpliga åtgärder.

Konstruktorn tar en Socket till anslutningen till servern.
`ListenerThread(Socket) : ListenerThread`

5.24.4. Subordinates
N/A

5.24.5. Dependencies
Tråden skapas och startas i Listener[L] som alltså måste existera.

5.24.6. Interfaces
En tråd för att hantera klientförfrågningar. Inflöde och utflöde av data sker via socketen.

5.24.7. Resources
N/A

5.24.8. References
N/A

5.24.9. Processing
Klienten mottar data från klienten via socketen och vidtar lämpliga åtgärder. Därefter skickas ett eventuellt svar till klienten.

5.24.10. Data
`clientSocket : Socket`
Socketen, det vill säga anslutningen till servern.

5.25 [DBC]
DBConn.

5.25.1. Type
Klass.

5.25.2. Purpose
Ett enhetligt objekt på servern som hanterar en databasanslutning och utför samtliga SQL-frågor.

Behövs för de URD-krav där klienten ska interagera med databasen, nämligen 3.1.4, 3.1.5, 3.1.6, 3.1.7 och 3.1.12.

5.25.3. Function
Skapar en anslutning till serverns databas. Innehåller metoder med SQL-frågor som används av andra komponenter i servern för att läsa och skriva data.

Klassen nås genom att anropa metoden getInstance som returnerar en instans av klassen enligt designmönstret singleton.

Metoder:
De diverse metoder som hanterar SQL-frågor skrivs efter behov. Dessa metoder ska returnera objekt vars datatyp specificeras nedan. Returtyperna delas in i tre stycken huvudsakliga kategorier efter hur det eftersökta datat är strukturerad:

1. Ett enstaka värde : `String`
2. En tupel ur databasen : `Map<String, String>`
3. En lista av tupler : `List<Map<String, String>>`

För övriga önskvärda datatyper, till exempel booleans för inloggningskontroll, kan returtyp definieras efter behov.

`getInstance() : DBConn`
Returnerar det enda objektet av klassen om det finns, annars skapas det först.
`validateLogin(String, String): boolean`
`getUserList(String) : ResultSet`
Får listan med användare innehållande förnamn och efternamn samt tillhörande roller.
`getNewToken(String) : String`
Genererar en ny token och lagrar den i databasen.

5.25.4. Subordinates
N/A

5.25.5. Dependencies
N/A

5.25.6. Interfaces

Klassen används av samtliga komponenter i servern som behöver skriva till eller läsa från databasen.

När en förfrågan som kräver autentisering skickas från en inloggad användare (klient med token), behöver Listener[L] anropa klassen med lämplig metod för att hämta relevant data för autentiseringen.

Samtliga Procedures[P] anropar metoder i klassen som skriver till eller läser från databasen med SQL-frågor och som returnerar relevant resultat.

Datatypen som returneras av SQL-frågemetoderna i klassen är relevant eftersom samma data analyseras i ServerOutputFormatter[SOF] för att konstruera JSON-strängar som ska skickas till en klient.

5.25.7. Resources
Serverhårdvara och MySQL-databas.

5.25.8. References
N/A

5.25.9. Processing
Klassen utnyttjar designmönstret *initialization-on-demand holder*, vilket är en trådsäker variant av singleton. Singleton är lämpligt eftersom det endast ska finnas en anslutning till databasen och eftersom objektets tillstånd inte kan påverkas av andra komponenter. Detta minimerar potentiella fallgropar med singletons. Trådsäkerhet är viktigt eftersom det kan hända att två trådar, exempelvis två ListenerThreads[LT], anropar getInstance samtidigt innan objektet är initialiserat.

Kodurklipp:
```
private static class Handler {
      public static final DBConn INSTANCE = new DBConn();
}

private DBConn() {
      // private constructor
}

public static DBConn getInstance() {
      return Handler.INSTANCE;
}
```

Diverse processing består av SQL-frågor och uppbyggnad av returdata.

5.25.10. Data
`conn : Connection`
Anslutningen till databasen.
`st : Statement`
En fråga till databasen.
`rs : ResultSet`
Svaret på en fråga till databas.

`dbUrl : String`
Sökvägen till databasen.
`dbUser : String`
Användarnamnet för den sökande.
`dbPassword : String`
Lösenordet för den sökande.

5.26 [P]

Procedure.

5.26.1 Type

Gränssnitt.

5.26.2 Purpose

Behövs för att hantera olika förfrågningar från klienten. Med hjälp av ett gränssnitt kan dessa följa en gemensam mall och vara på en generell form.

Relevant för URD-kraven 3.1.2, 3.1.4, 3.1.5, 3.1.7 och 3.1.10.

5.26.3 Function

Säkerställa att alla implementerande klasser innehåller metoden startProcedure.

Metoder:
`startProcedure() : void`

5.26.4 Subordinates

LoginProcedure[LoginP], PositionResponseProcedure[PResP], PositionRequestProcedure[PReqP], AddFavoriteProcedure[AFP] och DeleteFavoriteProcedure[DFP].

5.26.5 Dependecies

N/A

5.26.6 Interfaces

N/A

5.26.7 Resources

N/A

5.26.8 References

N/A

5.26.9 Processing

Metoden startProcedure startar en process som kan involvera kommunikation med databas, GCM och/eller andra klienter.

5.26.10 Data

N/A

5.27 [LoginP]

LoginProcedure.

5.27.1. Type

Klass.

5.27.2. Purpose

Ansvarar för att säkerställa att användarnamn och lösenord som användaren matat in vid inloggning är korrekta. Vid korrekt inmatade inloggningsuppgifter genererar klassen en token för att kunna bekräfta att en användare är inloggad.

Behövs för URD-kraven 3.1.4, 3.1.5, 3.1.7 och 3.1.12 eftersom det i dessa krävs att vi på ett säkert sätt kan skilja användarna åt.

5.27.3. Function

Tar emot användarnamn, lösenord och klientsocket. Validerar användarnamn och lösenord genom att skicka dessa till databasen som kontrollerar om de är korrekta.

Metoder:

```
startProcedure() : void
validateLoginRequest(String, String) : boolean
generateToken(String) : boolean
```

Får en ny token från DBConn[DBC] och skickar den till ServerOutputFormatter[SOF].

```
sendToken(String) : boolean
```

Skapar en JSON-sträng från token och anropar `send(String)`.

```
send(String) : void
```

Skickar en JSON-sträng till klienten.

Konstruktorn tar en Socket till anslutningen till servern samt användarnamn och lösenord.

```
LoginProcedure(Socket, String, String) : LoginProceduure
```

5.27.4. Subordinates

N/A

5.27.5. Dependencies

MySQL-databas med användarnamn och hashade lösenord. En klass som skapar en socket till klienten och en klass som skapar en anslutning till databasen.

5.27.6. Interfaces

Anropas av Listener[L] när en inloggningsbegäran görs från någon klient.

Inloggningsuppgifterna skickas till databasen genom att anropa DBConn[DBC]. Vid korrekta inloggningsuppgifter skickas en token till databasen via DBConn[DBC] och till klienten via klientsocket.

Vid en misslyckad inloggning, det vill säga om inloggningsuppgifterna inte matchar någon tupel i databasen, skickas endast ett felmeddelande som svar till klienten och ingen token skapas.

5.27.7. Resources
Serverhårdvara och MySQL-databas.

5.27.8. References
N/A

5.27.9. Processing
Startas av Listener[L] som skickar in klientsocket, användarnamn och lösenord till konstruktorn.

Metoden validateLoginRequest tar in användarnamn och lösenord. Lösenordet hashas och skickas tillsammans med användarnamnet till databasen. Om databasen svarar att inloggningsuppgifterna är korrekta genereras en token av metoden generateToken. Den genererade tokenen skickas sedan till databasen och klienten av metoden sendToken.

5.27.10. Data
`clientSocket : Socket`
Anslutningen till servern.
`user_id : String`
Användarnamn för den som vill logga in.
`password : String`
Lösenord för den som vill logga in.

5.28 [SP]
SearchProcedure.

5.28.1. Type
Klass.

5.28.2. Purpose
Möjliggör sökning efter användare.

Relevant för URD-krav 3.1.4 och 3.1.11 eftersom dessa krav berör sökning efter användare.

5.28.3. Function
Hämtar en lista över samtliga användare i databasen, formaterar listan och skickar den till klient över en Socket som fås då detta objekt skapas.

Metoder:
`startProcedure() : void`

Konstruktorn tar en Socket till kommunikation med servern.

`SearchProcedure(Socket) : SearchProcedure`

5.28.4. Subordinates
N/A

5.28.5. Dependencies
Det måste finnas en anslutning till en klient, det vill säga en Socket.

5.28.6. Interfaces
Ett objekt av klassen skapas och startas av Listener[L] när en förfrågning om en användarlista tas emot. Relevant data hämtas från databasen genom att anropa metoder i DBConn[DBC]. Därefter formateras datan genom att skapa ett ServerOutputFormatter[SOF]-objekt som sedan skickas till klienten.

5.28.7. Resources
N/A

5.28.8. References
N/A

5.28.9. Processing
N/A

5.28.10. Data
`clientSocket : Socket`
Anslutningen till servern.

5.29 [LogP]
LogProcedure.

5.29.1. Type
Klass.

5.29.2. Purpose
Klassen förmedlar information om vem som försökt lokalisera den frågande användaren.

Klassen är relevant för URD-kravet 3.1.7.

5.29.3. Function
Klassen gör en databasförfrågan och utvärderar svaret, varvid ett meddelande paketeras och sänds till den användare som initierat förfrågan.

Metoder:
`startProcedure() : void`

Konstruktorn tar en Socket för kommunkationen med servern och användarnamn för den användare som gör positionsförfrågan:

`LogProcedure(Socket, int) : LogProcedure`

5.29.4. Subordinates
N/A

5.29.5. Dependencies
Användning av klassen förutsätter att användaren är inloggad.

5.29.6. Interfaces
När kontrollen överförs till klassen tar denna emot en anslutning över en socket som är bekräftat aktiv och information om vilket användarnamn som förfrågan gäller. Klassen har sedan ansvaret att på denna anslutning skicka relevant information och sedan returnera utan returvärde varvid kontrollen återgår till ListenerThread[LT].

5.29.7. Resources
En databas tillgängliggjord genom klassen DBConn[DBC] och en strängformatterare, tillika meddelandepaketerare, i klassen ServerOutputFormatter[SOF]. Javaklassen PrintWriter behövs för att kunna skriva till socketen.

5.29.8. References
N/A

5.29.9. Processing
Klassen, som initialiseras i ListenerThread[LT] om användarens förfrågan visar att den vill använda denna funktion i servern, tar i sin konstruktor emot parametrar motsvarande en TCP-anslutning och ett användarnamn. Inledningsvis förbereds en SQL-fråga till databasen. Frågan körs sedan genom det globala objektet DBConn[DBC] som returnerar en lista av maps (List<Map<nyckel, värde>>) där varje nyckel motsvarar en kolumn i databasen och varje värde motsvarar kolumnens värde. Intressanta data att ta med är användarnamn (översatt till användarens riktiga namn) på den som sökt aktuell användare och vid vilken tidpunkt detta skett. Klassen kontrollerar nu om relevant data hittades i databasen och förbereder ett lämpligt svar till klienten genom att anropa ClientOutputFormatter[COF]. Slutligen skickas svaret på anslutningen varpå klassen returnerar.

Pseudokod:

```
LogProcedure(Socket conn, int uid) {
      query = DBConn.createStatement("loggdata", uid)
     resultat = DBConn.runQuery(query)
      om(resultat.length() == 0) svar = ClientOutputFormatter.format("Inga
svar hittades")
      annars svar = ClientOutputFormatter.format(resultat)
      conn.send(svar)
}
```

5.29.10. Data
`clientSocket : Socket`
En anslutning till den klient som efterfrågat informationen. Anslutningen är ett objekt.
`requesterID : int`
Ett unikt användarnamn som går att använda som referens vid sökning i databasen.

5.30 [PReqP]

PositionRequestProcedure.

5.30.1. Type

Klass.

5.30.2. Purpose

Behövs för URD-kraven 3.1.2, 3.1.4, 3.1.5, 3.1.7, 3.1.9 och 3.1.10 i vilka en klient efterfrågar en annan klients position.

5.30.3. Function

Hanterar inkommande positionsbegäran från en användare. Begär information från databasen för att kunna skicka en positionförfrågan via GCM till den användare vars position blivit efterfrågad.

Metoder:

```
startProcedure() : void
```

Konstruktorn tar ett registrerings-ID, en token till den sökta användaren och namn till den sökande användaren:

```
PositionRequestProcedure(String, String, String)
```

5.30.4. Subordinates

N/A

5.30.5. Dependencies

GCM, serverhårdvara och MySQL-databas.

5.30.6. Interfaces

Innefattar metoden startProcedure som skapar ett medelande som är anpassat för att sända via GCM och försöker sedan sända meddelandet.

5.30.7. Resources

GCM.

5.30.8. References

Kapitel 4.2.4.

5.30.9. Processing

Ett objekt av klassen skapas när en användare efterfrågar en annan användares position.

```
PositionRequestProcedure positionRequestProcedure =
     new PositionRequestProcedure(registrationId, token, name);
```

Koden ovan skapar ett meddelande för att sända via GCM. Innehållet i meddelandet är REQUEST_POSITION, token och name. Det är den efterfrågande användarens token som hämtats från databasen, eftersom den efterfrågade klienten ska kunna verifiera att förfrågan kommer från en giltig källa.

5.30.10. Data
```
sender : Sender
registrationId : String
token : String
name : String
key : String
```
API-nyckeln till GCM.
```
timeToLive : int
```
Tiden tills meddelandet tas bort från GCM-servern.

5.31 [PResP]
PositionResponseProcedure.

5.31.1. Type
Klass.

5.31.2. Purpose
Klassen har som uppgift att förmedla ett svar till användare som ställt en förfrågan om en annan användares position när detta svar inkommit till servern.

Behövs för URD-kraven 3.1.2, 3.1.4, 3.1.5, 3.1.7, 3.1.9 och 3.1.10 i vilka en klient efterfrågar en annan klients position.

5.31.3. Function
Klassen initialiseras under förutsättningen att en positionering ska förmedlas över en socket till en efterfrågande användare. Det första klassen gör är att översätta de beaconuppgifter som ges som inparametrar till ett rum genom att göra en databasförfrågan. När svar erhållits formateras en textsträng som sedan skickas på den socket som ges som inparameter.

Metoder:
```
startProcedure() : void
```

Konstruktorn tar en Socket för kommunkation med servern och positionen för den sökta användaren som en sträng:
```
PositionResponseProcedure(Socket, String) : PositionResponseProcedure
```

5.31.4. Subordinates
N/A

5.31.5. Dependencies
Innan klassen används krävs att servern via GCM lyckats förmedla en positionsförfrågan till avsedd målanvändare. Vidare krävs att målanvändaren är inloggad och är tillgänglig på ett datakommunikationsnätverk anslutet till Internet. Dessutom krävs att förfrågan utlöst förväntat beteende hos målanvändaren varefter denne initialiserat den anslutning som klassen i slutändan omhändertar. Innan klassen får kontrollen i servern fordras även att

informationen från den lokaliserade användaren genomgått lyckad autentisering samt att tupeln “förfrågande användare, målanvändare” går att associera med en aktiv anslutning i mappen pendingRequests.

5.31.6. Interfaces
Klassen får kontrollen när en ny anslutning upprättats över vilken data innehållande positioneringsinformation från en användare skickats. Anslutningen sker som ett svar på en från servern tidigare skickad förfrågan som skickats via GCM till målanvändaren. Då servern delegerar den nya inkommande anslutningen till en nystartad tråd sker några kontroller efter vilka ett objekt av klassen initialiseras med parametrarna conn, beaconID_maj och beaconID_min. Detta objekt ansvarar sedan för att översätta beaconID-parametrarna till ett rumsnamn som sedan ska skickas till den bekräftat aktiva anslutningen conn. När detta skett returnerar funktionen utan returvärde och kontrollen återgår till ListenerThread.

5.31.7. Resources
Klassen behöver via DBConn[DBC] tillgång till en databasserver och formateringsklassen ServerOutputFormatter[SOF] för att formatera den data som ska skickas i JSON-format till klienten. Javaklassen PrintWriter behövs för att kunna skriva till socketen.

5.31.8. References
N/A

5.31.9. Processing
Till en början skapas en databasförfrågan med syftet att översätta två beacon-parametrar till ett givet rum. Resulterar denna förfrågan i ett tomt svar finns inte den beacon som identifierats med i våra register och något har gått snett. Får vi ett svar paketeras svaret och skickas på socketen innan vi returnerar till ListenerThread.

Pseudokod:
```
RequestResponsePrecedure(Socket conn, int beaconID_maj, int beaconID_min) {
      query = DBConn.createStatement(“rumsöversättning”, beaconID_maj,
beaconID_min)
     resultat = DBConn.runQuery(query)
      om(resultat.length() == 0) {
svar = ServerOutputFormatter.format(“Misslyckad förfrågan”)
}
      annars svar = ServerOutputFormatter.format(resultat)
      conn.send(svar)
}
```

5.31.10. Data
`clientSocket : Socket`
En anslutning till den klient som efterfrågat informationen. Anslutningen är ett objekt.
`position : String`
Positionsresultatet av databasfrågan.

5.32 [AFP]
AddFavoriteProcedure.

5.32.1. Type
Klass.

5.32.2. Purpose
Möjliggör tilläggandet av favoriter för användare.

Klassen är relevant för URD-kraven 3.1.3, 3.1.12, 3.2.6 och 3.2.7 bland annat eftersom det ska gå snabbt att göra en sökning, framför allt efter användare man ofta sökt efter tidigare.

5.32.3. Function
Lägger till en favorit för en användare i serverns databas.

Metoder:
```
startProcedure() : void
```

5.32.4. Subordinates
N/A

5.32.5. Dependencies
N/A

5.32.6. Interfaces
Ett objekt av klassen skapas och startas av Listener[L] när en förfrågan om att lägga till en favorit tas emot.

Ett nytt favoritinlägg läggs till i databasen genom att anropa relevant metod i DBConn[DBC].

5.32.7. Resources
MySQL-databas.

5.32.8. References
N/A

5.32.9. Processing
N/A

5.32.10. Data
```
clientSocket : Socket
userID : String
favID : String
```

5.33 [DFP]
DeleteFavoriteProcedure.

5.33.1. Type
Klass.

5.33.2. Purpose
Möjliggör borttagande av favoriter för användare och är således relevant för URD-krav 3.1.3, 3.1.12, 3.2.6 och 3.2.7 eftersom det ska gå snabbt att göra en sökning, framför allt efter användare man ofta sökt efter tidigare.

5.33.3. Function
Tar bort favoriter för en användare i serverns databas. Tar emot en lista med favoriter som ska tas bort.

```
startProcedure() : void
```

5.33.4. Subordinates
N/A

5.33.5. Dependencies
N/A

5.33.6. Interfaces
Ett objekt av klassen skapas av Listener[L] då en förfrågan om att ta bort favoriter tas emot.

5.33.7. Resources
Databas.

5.33.8. References
N/A

5.33.9. Processing
N/A

5.33.10. Data
```
clientSocket : Socket
userID : String
favIDList : List<String>
```

6 Feasibility and Resource Estimates

6.1 Genomförbarhetsstudie

För att skapa oss en uppfattning om hur troligt det är att vi kommer klara av att utveckla applikationen har en genomförbarhetsstudie gjorts. Alla komponenter har då prioriterats efter hur kritiska de är för systemet, och en riskanalys har gjorts över de största riskerna under utvecklingen av applikationen. Utöver detta så har även en preliminär planering och en critical path gjorts för de olika komponenterna. Riskanalysen, planeringen och critical path-diagrammet återfinnes i dokumentet PPD-2 istället för i detta dokument, enligt instruktioner från examinator Karl Meinke på KTH Social.

Denna studie har gett en god överblick av vilka komponenter som är viktigast och måste prioriteras högst. Vi har även blivit medvetna om de största riskerna under utvecklingen av klient- och servermjukvaran, och vi har tagit fram förslag på hur dessa ska bemötas. Därför anser vi att vi har god möjlighet att lyckas med utvecklingen av applikationen. Om vi skulle stöta på problem så har vi möjlighet att enkelt prioritera de komponenter som är viktigast och som behövs för att åtminstone uppfylla minimikraven på applikationen.

6.2 Komponentprioritering

Samtliga komponenter har delats in i tre separata kategorier efter hur viktiga de är för arkitekturen. De allra viktigaste komponenterna är de som är nödvändiga för att andra komponenter ska kunna fungera. Exempel på sådana komponenter är de som möjliggör anslutning och kommunikation mellan klient och server, de som på serverns sida hanterar databasanslutning och de som på klientens sida tar kontakt med och avläser värden på beacons.

Följande prioritetsmodell används för att göra en diskret prioriteringslista. Samtliga komponenter uppdelas även efter deras tillhörighet till klient eller server.

- **Hög prioritet** – Dessa komponenter är grundstenar i systemet och måste således implementeras tidigt.
- **Medelprioritet** – Dessa komponenter måste implementeras efter komponenter med hög prioritet.
- **Låg prioritet** – Dessa komponenter måste implementeras efter komponenter med medelprioritet.

I tabellen nedan presenteras samtliga komponenters prioritet enligt prioriteringsmodellen ovan. Det finns även en kolumn i tabellen som visar need-värdet för de krav som fastställdes i URD-1 som är relevanta för respektive komponent. Det behöver inte nödvändigtvis finnas en korrelation mellan need och prioritet, men det är användbart för att kunna bilda en uppfattning om hur viktiga olika komponenter är.

Komponent (klient)	**Need**	**Prioritet**
FavoritesFile[FF]	Core, Standard	**Låg**
LoginDetails[LD]	Standard	**Låg**
FileManager[FM]	Core, Deluxe	**Låg**
BeaconID[BID]	Core, Standard	**Hög**
BeaconFactory[BF]	Core, Standard	**Hög**
User[U]	Core, Standard, Deluxe	**Medel**
UserList[UL]	Core, Deluxe	**Medel**
LogEntry[LE]	Deluxe	**Låg**
LogEntryList[LEL]	Deluxe	**Låg**
LoginInfo[LINFO]	Core	**Medel**
BeaconWhisperer[BW]	Core, Standard	**Hög**
ServerCon[SC]	Core, Standard, Deluxe	**Hög**
ConnectionHandle[CH]	Core, Standard, Deluxe	**Hög**
ClientOutputFormatter[COF]	Core, Standard, Deluxe	**Medel**
LoginActivity[LA]	Core, Standard	**Medel**
MainActivity[MA]	Core, Standard	**Medel**
FavoritesFragment[FFrag]	Core, Standard, Deluxe	**Låg**
SearchFragment[SF]	Core, Deluxe	**Medel**
LogbookFragment[LBF]	Core, Deluxe	**Låg**
SettingsActivity[SA]	Core, Deluxe	**Låg**
PersonActivity[PA]	Core, Standard	**Medel**

Komponent (server)	Need	Prioritet
ServerOutputFormatter[SOF]	Core, Standard, Deluxe	**Medel**
Listener[L]	Core, Standard, Deluxe	**Hög**
ListenerThread[LT]	Core, Standard, Deluxe	**Hög**
DBConn[DBC]	Core, Standard, Deluxe	**Hög**
Procedure[P]	Core, Standard, Deluxe	**Medel**
LoginProcedure[LP]	Core, Standard, Deluxe	**Medel**
SearchProcedure[SP]	Core, Deluxe	**Medel**
LogProcedure[LogP]	Deluxe	**Låg**
PositionRequestProcedure[PReqP]	Core, Standard, Deluxe	**Medel**
PositionResponseProcedure[PResP]	Core, Standard, Deluxe	**Medel**
AddFavoriteProcedure[AFP]	Core, Standard, Deluxe	**Låg**
DeleteFavoriteProcedure[DFP]	Core, Standard, Deluxe	**Låg**

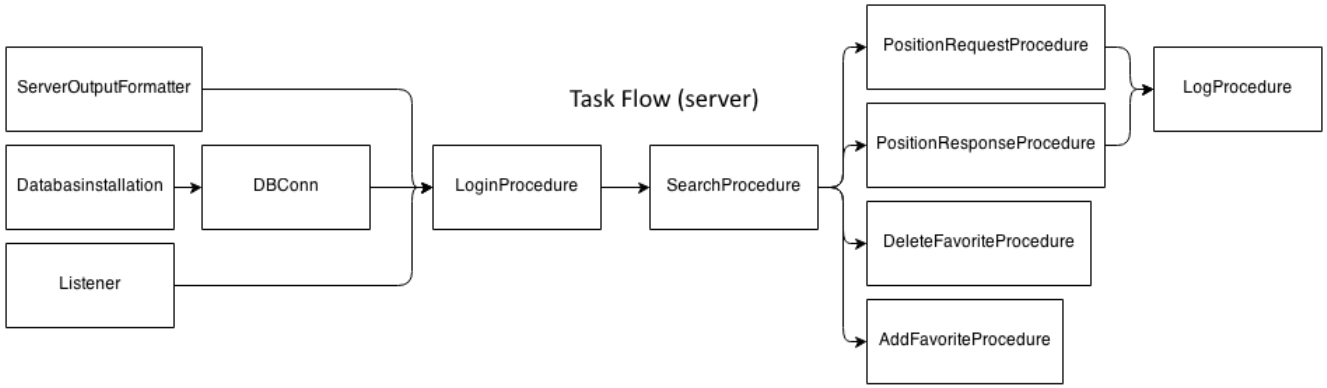
6.3 Project tasks

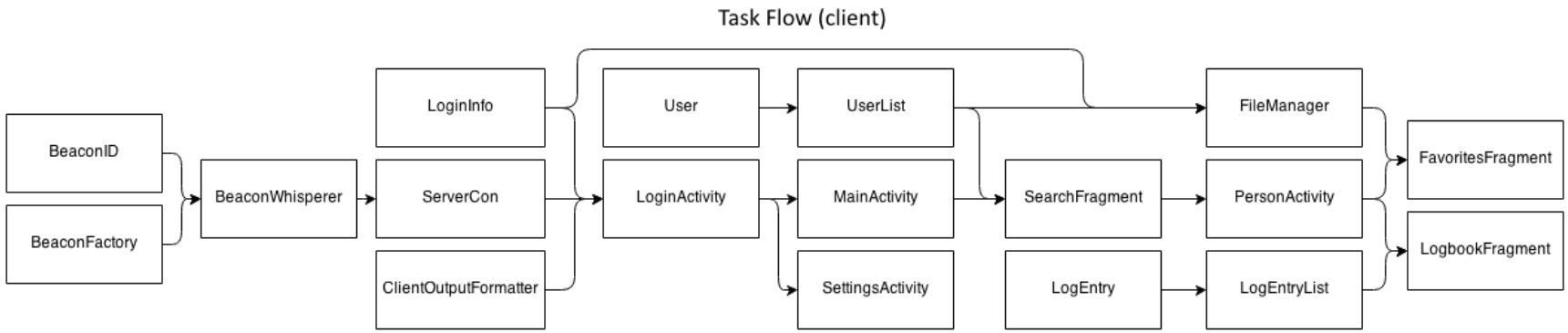
Project tasks, det vill säga konkreta uppgifter som ska göras inom en viss tidsram, är uppdelade efter komponenter. Varje komponent utgör en task i planeringen, med undantag av Procedure[P], FavoritesFile[FF] och LoginDetails[LD] som inte kräver något egentligt arbete. ConnectionHandle[CH] och ListenerThread[LT] är lokala klasser inom ServerCon[SC] respektive Listener[L], och ingår därför i deras komponent-tasks. Utöver komponenter så ses även installation av databasen och enhetstestning som tasks. Varje task som är en komponent avser design och kodning av tillhörande klasser och metoder samt enkel, informell testning för att säkerställa att den tänkta funktionaliteten uppfylls. Enhetstestning av kod utförs efter att alla andra tasks är utförda.

Planering av tasks påverkas av vilka komponenter som är viktigast för arkitekturen. Vissa komponenter är nödvändiga för att andra, mer kritiska komponenter ska fungera som tänkt, och påbörjas därför enligt planeringen i ett tidigare skede. För detta syfte har komponentprioriteringstabellen använts för att få en överblick.

Samtliga tasks, med undantag av enhetstestning, illustreras nedan i två *task flow-diagram*. Syftet med diagrammen är att ge en översiktlig uppfattning om i vilken ordning komponenterna bör påbörjas och färdigställas. Klient och server har separerats eftersom de tasks som tillhör respektive del kan utföras oberoende av varandra.

Diagrammen ska läsas från vänster till höger, och de visar i vilken ungefärlig ordning som tasks kommer att utföras efter vilka beroenden som finns mellan dem. Exempelvis går det att se att SearchProcedure är beroende av LoginProcedure i serverns diagram. Detta beroende existerar eftersom inloggning är en förutsättning för att sökningar ska kunna göras i den färdiga produkten, varför det är viktigt att inloggningen blir klar först. På samma vis är till exempel LoginActivity beroende av ClientOutputFormatter, ServerCon och LoginInfo eftersom dessa klasser behöver vara funktionella för att LoginActivity ska vara funktionell.


Task Flow (server)
ServerOutputFormatter
Databasinstallation
DBConn
Listener
LoginProcedure
SearchProcedure
PositionRequestProcedure
PositionResponseProcedure
DeleteFavoriteProcedure
AddFavoriteProcedure
LogProcedure


Task Flow (client)
BeaconID
BeaconFactory
BeaconWhisperer
LoginInfo
ServerCon
ClientOutputFormatter
User
LoginActivity
UserList
MainActivity
SettingsActivity
SearchFragment
LogEntry
FileManager
PersonActivity
LogEntryList
FavoritesFragment
LogbookFragment

6.4 Arbetsbelastning

En uppskattning har gjorts för hur stor arbetsinsats varje task kommer att kräva, och nedan illustreras arbetsbelastningen i ett diagram. Syftet med diagrammet är att ge en uppfattning om vilka tasks som kan tänkas leda till implementationssvårigheter och därmed hur pass mycket tid bör planeras för varje task.

Diagrammet inkluderar samtliga komponenter med undantag av Procedure[P], FavoritesFile[FF], LoginDetails[LD] och enhetstestning, eftersom dessa är redundanta eller opraktiska att göra belastningsuppskattningar för. ConnectionHandle[CH] och ListenerThread[LT] är lokala klasser inom ServerCon[SC] respektive Listener[L], och ingår därför i deras komponent-tasks.

Färgkodningen motsvarar komponentprioritering. Röd färg motsvarar hög prioritet, gul färg motsvarar medelprioritet och grön färg motsvarar låg prioritet. För tasks som saknar prioritering används färgen grå.

En detaljerad planering av när alla tasks ska vara färdigställda, vilka gruppmedlemmar som är ansvariga för vilka tasks och en uppskattning av tidsåtgången för varje task återfinnes i form av ett Gantt-diagram i dokumentet PPD-2 istället för i detta dokument, enligt instruktioner från examinator Karl Meinke på KTH Social.

6.5 Resource estimates

En grov uppskattning har gjorts över vilka resurser som kan tänkas behövas i utveckling och driften av klienten och servern. Med drift avses såväl användandet av slutprodukten (klient) som administration (server).

6.5.1 Utvecklingsmiljö

Utvecklingsmiljön för klienten består av Eclipse med versioner av ADT (Android Developer Tools) och Android SDK som stödjer utvecklingen av applikationer för Android 4.3 ellller högre. Dessa verktyg krävs för att kunna kompilera och köra applikationer på Android-enheter. För att kunna göra tester behövs enheter som kör version 4.3 eller högre av Android.

Utvecklingsmiljön för servern består av en utvecklingsmiljö för kodning och en klient för att ansluta till en avlägsen server genom SSH. Servern ska kunna bygga, kompilera och köra ett Java-program med hjälp av exempelvis Eclipse eller Apache Ant efter behov. Servern kör en MySQL-databas.

För övrigt ställs inga särskilda krav på utvecklingssystemens hårdvaruspecifikationer eller mjukvarukonfigurationer.

För utveckling av såväl klient som server används Git för versionshantering.

6.5.2 Driftmiljö

I driftmiljön krävs en server (hårdvara) som kan hantera kontinuerlig trafik från ett litet antal klienter. Servern nås genom SSH och använder för övrigt samma konfiguration som i utvecklingsmiljön.

För klientens driftmiljö behövs Android-enheter med stöd för BLE (Bluetooth 4.0) och Android 4.3. Ett antagande som vi gjort är att om en Android-enhet är tillräckligt kraftfull för att köra Android 4.3 så borde enheten även vara tillräckligt kraftfull för att kunna köra applikationen utan prestandarelaterade problem.

Inga övriga specificerade resurser krävs för driftmiljön.

7 User Requirements vs Components Traceability Matrix

Notera att URD-kraven 3.2.2, 3.2.3, 3.2.5, 3.2.8 och 3.2.9 inte finns med i matrisen på grund av deras ofunktionella natur som har mycket lite att göra med koden.

	3.1.1	3.1.2	3.1.3	3.1.4	3.1.5	3.1.6	3.1.7	3.1.8	3.1.9	3.1.10	3.1.11	3.1.12	3.2.1	3.2.4	3.2.6	3.2.7
5.1				x											x	x
5.2																x
5.3			x	x							x		x			
5.4	x	x			x			x					x			
5.5	x	x			x			x					x			
5.6				x	x	x	x			x	x					
5.7				x						x	x					
5.8							x									
5.9							x									
5.10						x										
5.11	x				x			x					x			
5.12				x	x	x	x									
5.13				x	x	x	x									
5.14				x	x	x	x									
5.15			x													x
5.16			x													x
5.17		x										x		x		x
5.18			x	x						x	x			x		
5.19			x			x	x					x				

	3.1.1	3.1.2	3.1.3	3.1.4	3.1.5	3.1.6	3.1.7	3.1.8	3.1.9	3.1.10	3.1.11	3.1.12	3.2.1	3.2.4	3.2.6	3.2.7
5.20			x						x							
5.21			x		x											x
5.22				x	x		x					x				
5.23				x	x		x					x				
5.24				x	x		x					x				
5.25				x	x	x	x					x				
5.26		x		x	x		x			x						
5.27				x	x		x					x				
5.28				x							x					
5.29							x									
5.30		x		x	x		x		x	x						
5.31		x		x	x		x		x	x						
5.32			x									x			x	x
5.33			x									x			x	x