# Dynamisk modulhantering i Java med OSGi-serviceplattformen

**Robin Rosenberg**

Kandidatuppsats i Datavetenskap
Åbo Akademi
Handledare: Andreas Dahlin, Mats Aspnäs
Tekniska fakulteten
Datavetenskap
Mars 2010

# Innehåll

**1 Inledning** **2**
1.1 Bakgrund . . . . . 2
1.2 Modularitet, inkapsling och komplexitet . . . . . 3
1.3 Distribution av program . . . . . 4
1.4 Avgränsningar . . . . . 4

**2 Serviceplattformen OSGi** **4**
2.1 OSGi:s grundbegrepp . . . . . 4
2.1.1 OSGi-ordlista . . . . . 5
2.1.2 Modulhierarki i Eclipse . . . . . 6
2.2 Moduler . . . . . 6
2.2.1 Manifestet . . . . . 6
2.2.2 Fragmentmoduler . . . . . 8
2.2.3 Utökningsmoduler . . . . . 8
2.3 Versioner . . . . . 8
2.3.1 Validering av moduler . . . . . 9
2.4 Tjänster . . . . . 9
2.5 Livscykelhantering . . . . . 9
2.6 Proxytjänster . . . . . 10
2.7 Utökningar . . . . . 10
2.8 Startnivåer . . . . . 10
2.9 Administration . . . . . 11
2.10 Säkerhet . . . . . 12
2.10.1 Systemet med publika nycklar . . . . . 12
2.11 Implementationer . . . . . 13
2.12 Program gjorda i andra språk än Java . . . . . 13
2.13 Kommunikation mellan moduler . . . . . 14
2.13.1 Fjärrtjänster . . . . . 14
2.14 Tjänstespåraren och deklarativa tjänster . . . . . 15
2.15 Ramverks-API:et . . . . . 15
2.16 Andra JVM-språk än Java . . . . . 15
2.17 Problem och nackdelar med OSGi . . . . . 16

**3 Jämförelse med andra modeller** **17**
3.1 Modulnivå . . . . . 17
3.1.1 Java Enterprise Beans . . . . . 18
3.1.2 OLE, COM, ActiveX och DCOM . . . . . 18
3.1.3 CORBA . . . . . 19
3.1.4 Web Services . . . . . 19
3.1.5 Dotnet . . . . . 20
3.2 Operativsystem . . . . . 21
3.2.1 Linux . . . . . 21
3.2.2 Windows . . . . . 21

**4 Avslutning** **21**

**Referenser** **22**

# 1 Inledning

Vartefter mjukvarusystem växer till sig blir behovet av att separera komponenterna större och större. Utan kontroll och begränsning av beroendena mellan delarna i systemet ökar komplexiteten lavinartat och möjligheten att förstå vad systemet gör och att underhålla det minskar krafttigt. Med $n$ komponenter som alla kommunicerar med alla har man $n{\cdot}(n-1)$ kommunikationsvägar. Med ett dussin komponenter har man redan där över etthunda kopplingar och möjligheten att förstå systemet därefter.

I konstruktionen av system strävar man därför att hålla nere antalet kopplingar för att göra systemet begripligt. Ett problem är att ambitionen varierar och utan tvingande styrning ökar antalet kopplingar lätt. Många programmeringspråk erbjuder visst stöd för att öka modulariteten, men modulerna är väldigt fingranulära och i körtidsmiljön finns i regel väldigt lite extra stöd för att tvinga fram upprätthållandet av väldefinierade gränssnitt.

## 1.1 Bakgrund

Programmeringsspråket Java [15] och den virtuella maskin, Java Virtual Machine (JVM) [26], som hör till är ett populärt programmeringsspråk. Ursprungligen togs det fram för att stöda utvecklingen av små inbyggda system som fjärrkontroller, digital-TV-boxar och liknande elektronisk utrustning [14]. Inbyggda (även inbäddade) system har sina egen problem. Ett är att de har väldigt lite minne, det är svårt eller omöjligt att byta ut programmen i dem och toleransen för programfel hos användarna är nästan obefintlig. Java erbjuder en betydligt stabilare utvecklingsplattform än t.ex. C, C++, eller Assembler och dessutom portabilitet, dvs samma program kan köras på olika processorer, vilket är en fördel eftersom tillverkare av den typen av utrustning gärna byter ut hårdvaran mot billigare om det går.

Huruvida Java alltid är bättre än t.ex. ingår i en evig strid, s.k language wars. Det är dock klart att både för och nackdelar finns. Javas minneshantering i form av automatisk skräpsamling är ett område som erbjuder fördelar i form av förenklad minneshantering eller vanligare GC (garbage collection), men det finns också en kostnad if form av ökat minnesbehov och minskad prestanda [17]. I en studie [31] där 38 långt framskridna studerande själv får välja bland C, C++ och Java för att lösa ett mindre, icke-trivialt, problem, visar det sig att det finns ytterst liten korrelation mellan programmeringsspråk och exekveringstid. Även minnesförbrukning i lösningarna var i många fall mindre för Java-program än för C/C++-program, trots att den version av Java (Sun Java 1.2) som användes hade en så stor overhead som 20 megabyte. Om man beaktar att Java, sedan studien gjordes 1999, har utvecklat enormt vad gäller prestanda och även minnesutnyttjande kan man anta att Java, med hotspot-kompliatorn, bättre skräpsamling (Garbage Collection) och minnesmappade runtime-bibliotek, idag står sig betydligt bättre i jämförelse med C och C++. Slutsatsen Prechelt drar i studien är att skillnader i prestanda och minnesutnyttjande beror mera på vem som skriver programmet, än vilket programmeringsspråk som används. EN liknande studie med flera språk är dokumenterad i [32]. Se även [30] om varför det är ont om jämförande studier av detta slag.

En, enligt webbplatsen själv, icke helt vetenskaplig, men mera uppdaterad, jämförelse, mellan programmeringsspråk finns under namnet The Computer

Language Benchmarks Game[1]. Bänktester är mycket vanskliga, men överlag ger de resultaten att Java står sig relativt väl jämfört med C och C++.

Den andra aspekten beträffande fördelar av Java vs C/C++ handlar om verktyg. Floran av bra verktyg för Java är betydligt större än för C/C++ och de utvecklas mycket snabbt. Det finns bra verktyg för C och C++, men de är färre utvecklingen på verktygsfronten för dessa har stagnerat i jämförelsen med utvecklingen av verktyg för Java.

## 1.2 Modularitet, inkapsling och komplexitet

Ett typisk system idag kan delas upp i dessa komplexitetsnivåer i stigande storleksordning, från den minsta komponenten, till den största sammansättningen av komponenter.

| Procedur, Variabel | Den minsta refererbara enheten |
|---|---|
| Klass | En (liten) grupp av variabler och procedurer |
| Paket | En (liten) grupp av relaterade klasser |
| Modul | Ett antal relaterade paket |
| Delsystem | En relativt självständig grupp av moduler |
| System | Ett relativt självständig kombination av moduler |

Tabell 1: Olika lager i ett datorsystem

De exakta termerna och hur många nivåer som finns varierar mellan programmeringsspråk och metodik. T.ex. motsvarar en klass i Java närmast en Modula2-modul, medan en modul i Java ofta avser en större enhet. Det viktiga här är att det är en hierarkisk uppdelning där nästa nivå är "större" än den föregående. Genom att definiera gränssnitt på varje nivå, som är ett subset av de interna gränssnitten på den nivån, minskar man antalet frihetsgrader och kan bibehålla underhållbarheten i systemet när vissa aspekter kan ändra utan att allt runtomkring behöver påverkas. T.ex. har en Java-klass ett antal variabler och metoder (procedurer, funktioner), men bara vissa av dessa är tillgängliga för andra klasser. Dessa definierar klassens gränssnitt. Det som inte ingår i gränssnittet kan allså implementatören ändra fritt så länge effekten av förändringen inte är till men för den som använder gränssnittet.

Inom ett paket kan alla klasser använda alla andra klassers gränssnitt, men bara vissa klasser är tillgängliga för klasser i andra paket. På den nivån slutar i princip den abstraktionsnivå som Java (liksom många andra programmeringsspråk) erbjuder. Varje gång man skalar bort en del av systemet på detta sätt minskar man antalet sätt man kan kombinera systemets komponenter och framförallt de felaktiga och oavsiktliga sätten, som antagligen är fler än de korrekta sätten. Det förefaller sig alltså naturligt att man kunde begränsa åtkomsten även på högre nivå, som till paket. Sådana lösningar finns, vilket denna uppsats skall gå in på.

För den som bygger fysiska system är detta med inkapsling inte svårt att förstå eftersom det oftast finns en direkt kostnad associerad med att göra något tillgängligt, som t.ex. att lägga in en extra kopplingsdosa, medan denna direkta kostnad inte finns i mjukvara. Att skapa en privat eller publik metod i Java är precis lika enkelt. Riskerna med att ändra existerande gränssnitt är också ibland mera uppenbara i den fysiska miljön. Att ändra en spänning från 5 volt till 220

volt kan vara förenat med livsfara och alltså exponerar man inte en spänningsförande komponent om man inte måste. Risken med att börja returnera lite större värden från en funktion i mjuän tidigare är inte heller uppenbart [9, sekt 2.1].

### 1.3 Distribution av program

De första generationerna inbäddade system kom med programmen i ROM (Read Only Memory) eller liknande fasta minnen vars innehåll definieras en gång och inte kan ändras, eller svårligen kan ändras. Distribution av program skedde vid tillverkningen av apparaterna. Det innebar att programmen fanns i direkt i minnet när strömmen slogs på och även laddningsmekanismen är väldigt enkel. Ofta skedde ingen "laddning" utan programmet kördes direkt från det minne det låg i.

Idag innehåller dessa apparater mycket komplex programvara och därmed är risken för fel större, dvs i praktiken oundviklig. Det måste också vara möjligt att uppdatera programmen, ibland t.om. medan programmen körs eftersom en total omstart innebär en driftstörning.

I Java är minsta programenheten en klassfil. Det är en förpackad uppsättning instruktioner för en JVM och metadata kring dessa för att definiera namn på klasser, metoder och argument [15, s334]. Ett enkelt program som är inbyggt i JVM:en ser till att de klasser som hör till den körtidsmiljö kan laddas. Däri ingår mera avancerade klassladdare som kan analysera den kod som laddas och ladda från andra ställen är lokal disk [15, s313ff].

### 1.4 Avgränsningar

OSGi är ett relativt enkelt ramverk, men antalet tjänster som finns är ändå för stort för att få plats i en kandidatavhandling. Av den anledningen täcker detta inte administrationstjänsterna eller de flesta specifikationer som finns i "compendium"-delen av OSGi-specifikationen.

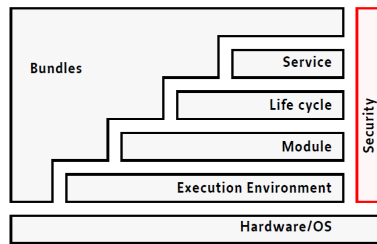
## 2 Serviceplattformen OSGi

OSGi är ett ramverk och en plattform för att köra Javaprogram i en strikt kontrollerad moduluppbyggd miljö.

### 2.1 OSGi:s grundbegrepp

För att förklara hur OSGi fungerar måste jag definiera några begrepp och dess struktur. OSGi har en grov struktur enligt följande [12, s 2].

Detta sätt att bygga ett ramverk lager på lager är rätt typiskt och inte någon revolution. Säkerhetslagret lägger till nya funktioner i varje lager och är därför uppritat på tvären jämfört med övriga lager för att betona att det är en och samma säkerhetsmodell som gäller för alla lager. Det är också ett valbart laget i och med att OSGi-specifikationen inte kräver att säkerhetslaget är implementerat.




Figur 1: OSGi:s olika lager [12, s2]

### 2.1.1 OSGi-ordlista

modul
: (bundle) En OSGi-term för ett antal klasser som utgör en modul tillsammans med metadata som beskriver modulen. OSGi använder inte termen module, men det är egentligen ingen skillnad och termen modul gör sig bättre på svenska.

livscykel
: (life cycle) De olika tillstånd en modul kan befinna sig i såsom installerad, definierad (resolved), startande, aktiv, stoppande och avinstallerad.

exekveringsmiljö
: (execution environment) Den Java-runtime som används, dess version och ingående API:er och deras implementationer.

händelse
: (event) en mekanism för moduler att berätta för andra komponenter att något hänt och låta dessa agera.

klass
: (class) Den minsta praktiskt hanterbara enheten programkod som det finns stöd för att hantera i standard-java. En klass kan ha metoder som associeras med instanser av klassen, eller anropas direkt. Lagring av data kan ske på klass eller instansnivå. En klass kan jämföras med den minsta tänkbara modulen.

klassladdare
: (class loader) Kod som hämtar in Java-klasser och gör dem tillgängliga för den virtuella maskinen. Det finns många olika klassladdare i Java för att hämta kod enligt olika regler, från olika ställen och för att kontrollera koden innan den laddas. Varje klassladdare ansvarar för sina klasser och samma klass kan laddas av olika klassladdare. I regel använder en klass i första hand “sin egen” klassladdare för när den behöver ladda andra klasser.

paket
: (package) Java-paket i en gemensam namnrymd. T.ex ingår java.util.Map och java.util.List i samma paket. Java-paket är inte hierarkiska så java.util.concurrent.ConcurrentHashMap ingår inte i samma paket som java.util.List.

tjänst
: (service) en procedur som kan anropas för att uträtta arbete eller svara på en fråga. I OSGi innebär det samma sak som ett instantierat objekt som kan svara enligt ett definierat protokoll via vanliga Java-anrop och kan hittas via register tjänsten.

register (registry) en katalog över tjänster och var de finns lokaliserade.

produktegenskap (feature) en gruppering av moduler som representerar en sammanhängande grupp av funktionalitet.

produkt (product) en gruppering av produktegenskaper till en färdig exekverbar enhet

### 2.1.2 Modulhierarki i Eclipse

I 1.2 fanns en någorlunda generisk illustration av lagren i ett mjuvarusystem. I OSGi ser detta schema ut på detta sätt

| (Procedur, Variabel) | Den minsta refererbara enheten |
|---|---|
| Klass | En (liten) grupp av variabler och procedurer |
| Paket | En (liten) grupp av relaterade klasser |
| Modul | Ett antal relaterade paket |
| Produktegenskap (feature) | En logisk gruppering av sammanhängande funktionalitet |
| Produkt | En komplett sammanställning av features och moduler |

Tabell 2: lager i OSGi/Eclipse

## 2.2 Moduler

Den första nivån av OSGi som man använder gäller moduler [12, kap 3].

OSGi-ramverket är ett Java-program som kör andra Java-program. Moduler registreras i ramverket som sedan aktiverar och startar moduler enligt vad som är konfigurerat. I vissa fall, som t.ex. Eclipse finns en mekanism som söker igenom ett antal kataloger vid uppstart och därefter kan även nätplatser sökas av för att hitta nya moduler och utökningar av moduler.

I OSGi används registret för att hitta buntar. Det går att söka efter moduler och tjänster genom att fråga enbart efter ett gränssnitt (Java interface), eller att ange ett villkor för att begränsa sökningen för det fall att flera buntar erbjuder olika implementationer av samma gränssnitt. En modul kommer inte åt innehåll i en annan modul om det inte är explicit tillåtet.

### 2.2.1 Manifestet

Förutom att en modul brukar innehålla Java-klasser och resurser finns där ett manifest som anger kontraktet mellan modulen och omvärlden.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Toast Emergency
Bundle-SymbolicName: org.equinoxosgi.toast.client.emergency
Bundle-Version: 1.0.0.20100327
Bundle-RequiredExecutionEnvironment: J2SE-1.4
Import-Package: org.eclipse.core.runtime.jobs,
 org.equinoxosgi.toast.core;version="[1.0.0,2.0.0)",
 org.equinoxosgi.toast.core.channel.sender;version="[1.0.0,2.0.0)",
 org.equinoxosgi.toast.core.emergency;version="[1.0.0,2.0.0)",
 org.equinoxosgi.toast.dev.airbag;version="[1.0.0,2.0.0)",
 org.equinoxosgi.toast.dev.gps;version="[1.0.0,2.0.0)"
Require-Bundle:
 org.eclipse.equinox.common;bundle-version="3.5.0"
Export-Package:
 org.equinoxosgi.toast.client.emergency;version="1.0.0",
org.equinoxosgi.toast.internal.client.emergency;x-internal:=true,
Service-Component: OSGI-INF/component.xml
```

Figur 2: Exempel på ett Manifest

Manifestet ovan är ett exempel från övningarna i [27]. Delarna i fet stil är de attribut som OSGi beaktar. Resten är dokumentation. Exemplet är inte heller komplett, i det att det finns många fler attribut.

`Bundle-SymbolicName` är det officiella namnet på modulen.

`Bundle-Version` är som antyds, versionsnumret. major=1, minor=0, micro=0, qualifier 20100327. Qualifierdelen i detta fall har formen av ett datum, vilket är konventionen för Eclipse-moduler, men vad OSGi anbelangar är det en textsträng. Versionsnummer har specifik tolkning i OSGi, vilket behandlas i 2.3.

`Import-Package` anger vilka java-paket (inte moduler just här) som denna modul behöver och vilka versioner av dessa paket som är acceptabla. I detta fall anges att alla med samma major-version duger.

`Require-Bundle` anger ett beroende till en annan modul i sin helhet. I detta fall är det OSGi-implementation i Eclipse 3.5 som pekas ut.

`Export-Package` anger att två av Java-paketen i denna modul skall vara tillgängliga för andra. Den första exporteras som verson 1.0, vilket innebär att andra moduler som behöver detta paket och anger versionkrav som matchar 1.0 så kan den använda denna modul. Den andra delen exporterar ett paket som inte egenligen ingår i API:et för denna modul, men i en situation där API:erna utvecklas ger det en viss flexibilitet och x-internal resulterar i att den som försöker använda denna modul får en varning i OSGi-medvetna utvecklingsverktyg (t.ex. Eclipse), men eftersom gränssnittet är exporterat så kan det ändå användas. Ytterligare några direktiv finns för att hantera synlighet, men dessa tas inte upp här, men t.ex. så kan man exportera ett paket bara till vissa moduler.

`Service-Component` pekar ut en fil i modulen som definierar hur denna modul ser ut beträffande tjänster, både i form av vilka tjänster den publicerar och konsumerar. Se 2.14.

### 2.2.2 Fragmentmoduler

För speciella ändamål som enhetstestning, kan man vilka gå runt de vanliga reglerna för åtkomst av implementationsdetaljer detaljer. OSGi stöder detta genom erbjuda fragmentmoduler [12, sekt 3.13]. En fragmentmodul kan läggas på en annan modul och blir effektivt ett med den modul den är anpassad för.

### 2.2.3 Utökningsmoduler

Om man behöver utöka OSGi-ramverket på låg nivå så kan man definiera utökningsmoduler, (extension bundles) [12, sekt 3.14]. Dessa moduler följer inte OSGi:s normala regler för klassladdning utan läggs in i Javas virtuella maskins s.k bootclasspath. Det kan vara utökningar i Java-runtime. OSGi-specifikationen nämner java.sql som exempel. Java.sql ingår i J2SE, men i en J2ME-miljö behöver paketet inte ingå och då kan man lägga till en implementation på detta sätt. En utökningsmodul ingår därefter i klassökvägen för alla moduler i systemet utan att dessa moduler importerar dem.

## 2.3 Versioner

Grundläggande villkor för att hitta en modul och dess ingående paket är modulnamn och version. OSGi är väldigt specifikt vad gäller tolkningen av versionsnummer. Både moduler och paket har versionsnummer i OSGi och man kan definiera beroenden till ett paket direkt, eller genom att referera till dess modul. Ett pakets versionummer behöver inte heller vara detsamma som modulens versionsnummer.

Ett versionsnummer består av flera delar: “major”, “minor”, “micro” och “qualifier”, representerande olika grader av olika kompatibilitet. OSGi definierar en tolkning av dessa versionsnummer på följande sätt, t.ex. 1.2.3.

En ändring i den första delen (major), representerar en inkompatibel ändring, dvs en klient (användare av ett gränssnitt) som känner till den äldre versionen (lägre nummer) kan inte kommunicera med implementationer av det nyare gränssnittet.

Nästa del av versionsnumret är “minor”, som representerar en kompatibel ändring, dvs klienter som känner till det äldre gränssnittet kan kommunicera med implementationer av det nyare gränssnittet. Kompatibiliteten kan gälla tvärtom också, men behöver inte göra det och OSGi har inget stöd för detta. En klient bör alltså definiera minimikravet på minor-versionen.

Den minsta sifferdelen är “micro” som representerar en fullt kompatibel ändring på gränssnittsnivå och i princip också beträffande funktionen, men man kan ha rättat ett fel.

Den fjärde delen (“qualifier”) är en valbar sträng som kan betyda olika saker för olika moduler. En typisk användning är att man lägger in en tidpunkt för att kunna skilja på olika versioner under utveckling. För officiella utgåvor utgår väljer man ibland att ha en tom kvalificerare. För att välja den nyaste modulen, vilket OSGi gör normalt, så jämförs kvalificeraren som en sträng om versionerna i övrigt är lika. Alla andra delar jämförs som numeriska heltalsvärden. Om en del av en version utelämnas så tolkas den som 0, förutom för kvaliceraren som då tolkas om tom sträng. Ett utelämnat versionnummere tolkas som “0.0.0”.

Några exempel på jämförelse av versioner.

$$1.2.0 \geqq 1.1.9$$

$$1.2.1 \geqq 1.2.0$$

$$1.2.0.z \geqq 1.2.0$$

$$1.2.0.z \geqq 1.2.0.a$$

$$1.2.0 \geqq 1.1.9$$

$$2.2.0.a \geqq 1.3.9.z$$

När man (en annan modul) begär en referens till en modul anger man vilken version man vill ha, med givna avgränsningar. Om man när man begär en viss version utelämnar en del så betraktas det som att man angivit 0 för ett numeriskt värde eller tom sträng för kvalificeraren, dvs 1.2 = 1.2.0 och 1.2.3 = 1.2.3.*"tom sträng"* [12, 3.5.3, 3.6.2]. Detta är inte helt kompatibelt med andra konventioner för att namnge versioner.

Förutom version kan komplexa villkor på godtyckliga extra attribut använda som villkor i komplexa uttryck för att välja ut vilken implementation som skall väljas. Man kan t.ex. ha olika moduler för olika operativsystem eller språk.

### 2.3.1 Validering av moduler

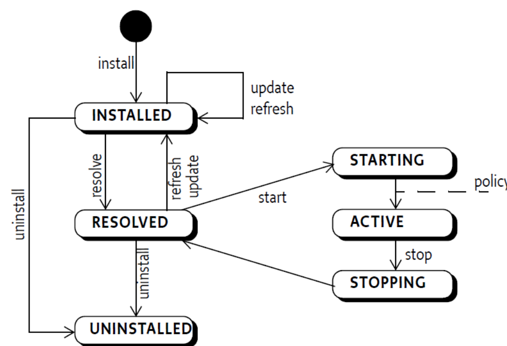
Många attribut i deklarationen av manifestet till en modul valideras för att se att den är korrekt definierad. Ett ogiltigt manifest får som konsekvens att modulen inte kan installeras och detta kan leda till andra moduler som behöver denna inte kan installeras.

## 2.4 Tjänster

Publicering av tjänster kan liknas vid deklarationen av moduler. Tjänster publiceras av aktiva moduler och refereras till av aktiva moduler. I grunden är en tjänst något så enkelt som en instans av en Javaklass som registrerats i tjänsteregistret [12, sek 5.2].

## 2.5 Livscykelhantering

Moduler i OSGi har en definierad livscykel som anger vad som skall ske när en modul laddas, aktiveras, deaktiveras, uppdateras, deaktiveras och avinstalleras.




Figur 3: En moduls livscykel

## 2.6 Proxytjänster

OSGi ger möjligheter att kapsla in tjänster och skapa proxytjänster för andra tjänster. En proxytjänst träder in i den ursprungliga tjänstens plats och modifierar eller ersätter tjänsten.

Att skapa proxytjänster är förenat med vissa problem eftersom proxy skapas efter tjänsten som döljs och då kan man inte vara säker på att alla anrop går via proxytjänsten [12], 12.3.1.

## 2.7 Utökningar

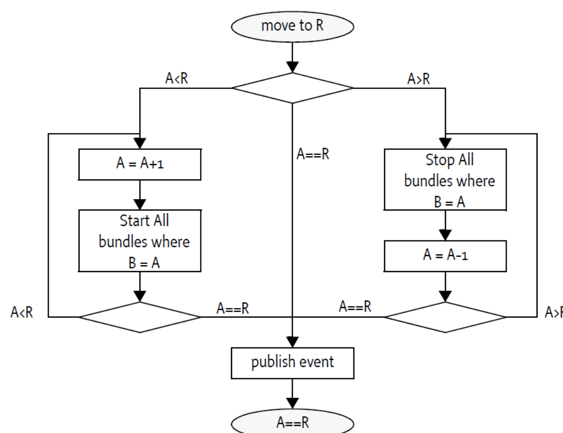
Eftersom Eclipse är en så dominerande del av OSGi kan det vara värt att notera att denna användbara del inte är en del av OSGi. Utökningar (extensions) är ett mekanism för moduler att bjuda in andra moduler för att erbjuda en utökad och integrerad funktionaliet [27, s275-293]. Ett mycket synligt exempel är de menyer som finns. Att det finns menyer är något som en modul i Eclipse ansvarar för, men den exponerar ett antal utökningspunkter där ytterligare moduler kan koppla in sina egna undermenyer eller menyelement. Andra exempel är utökningar för att koppla in versionshanterare i utvecklingsverktyg så att den som erbjuder en utvecklingsmiljö för t.ex. Java inte behöver implementera gränssnitt mot någon enda versionshanterare. Det “enda” som behövs är att ett antal utökningspunkter definieras så att tredje part kan erbjuda moduler för detta. I fallet med t.ex. versionhantering finns det en standard för vilka utökningingar som förväntas att en versionshanterarmodul (team provider) implementerar.

I viss mån liknar detta dynamiska tjänstedeklarationer och det är inte alltid uppenbart vilken mekanism man skall använda, men om man skriver en plugin som skall passa in redan existerande pluginer så finns det kanske bara ett sätt, dvs att implementera pluginen som utökningar till en eller flera utökningspunkter [27, s290-292].

## 2.8 Startnivåer

Administratören av ett OSGi-system kan ange en startnivå (“start level”). Om OSGi-implementationen stöder detta så kommer moduler att aktiveras i tur och ordning beroende på deras startnivå med upp till den nivå man angett att skall gälla för hela systemet. Om administratören säger att denne vill tar upp

systemet till nivå 3, kommer OSGi att först starta modulerna på nivå 1, sedan de på nivå 2 och slutligen på nivå 3. Startnivån för systemet kan sedan ändras och OSGi kommer i tur och ordning att starta eller stoppa nya moduler, en nivå åt gången. Moduler med samma startnivå startas i en sinsemellan odefinierad ordning. Antal möjliga nivåer är stor, upp till $2^{31} - 1$. [12, sekt 8]




Figur 4: Hantering av startnivåer [12, s238]

## 2.9 Administration

Behörigheter för kod kan specificeras i detalj ner på metodnivå om så behövs. Autentiseringen sker baserat på hur koden är signerad. M.a.o. är det ett fullt tänkbart scenario att kod från helt olika källor installeras i samma system. Det är i OSGi-sammanhang samma virtuella maskin. Signerade programpaket är inget nytt i sig. Linuxdistributioner som t.ex. RedHat och många andra har använt signerade programpaket under lång tid. Om ett paket inte är signerat med en nyckel som systemet känner till, dvs motsvarande publika nyckel är godkänd av systemet, kommer paketet inte att kunna installeras. Det fungerar framförallt som ett skydd mot trojanska hästar, men även mot att man installerar paket som skadats av misstag. Väl installerat sker ingen behörighetskontroll baserat på signaturen utan det är lokala inställningar i det aktuella systemet som styr detta utan hänsyn till signaturer.

OSGi:s detaljerade behörigheter baserat på signatur gör att man kan använda osäkra kanaler för programdistribution, eftersom obehörig kod, inte kan registreras eller användas..

Huvudproblemet består i att bestämma vem man kan lita på och vilka befogenheter dessa har.

- Vem får installera kod
- Vems kod får köras
- Vilken kod får göra vad
  - Tillgång till tjänster
  - Tillgång till resurser

- Hur identifierar man någon i systemet
- Indirekt förtroende, ex. jag litar på X därför att Y säger att X är ok.

[12, sek 9]

OSGi stöder även dynamiska villkor vid kontroll av behörighet, dvs en funktion evalueras och avgör om ett villkor är uppfyllt, t.ex. att en telefon ringer eller en knapp på telefonen är intryckt.

## 2.10 Säkerhet

OSGi är tänkt att fungera som en plattform för komponenter från olika källor, inklusive helt obereoende leverantörer. För att kunna garantera att dessa komponenter bara använder resurser de är behöriga till kan OSGi konfigureras på ett antal olika sätt för att begränsa vilka moduler som får anropa vilka moduler, vilken kod som får installeras etc.

I många installationer har man koll på vilka moduler som installeras utan OSGi:s hjälp och därför är säkerhetslagret valfritt för den som implementererar OSGi. Omvänt måste den som vill ha säkerhetslagret alltså välja en OSGi-implementation där detta ingår.

OSGi:s säkerhetsmodell ersätter inte, utan kompletterar den modell som finns i Java. Autentisering av kod innebär att ramverket avgör om en modul kan användas baserat på dess plats eller vem som signerat den.

Det enklaste scenariot är att man anger att klasser i ett visst paket får använda vissa resurser, t.ex. filer i en given katalog, eller får använda vissa klasser eller metoder. Detta anges av administören av systemet.

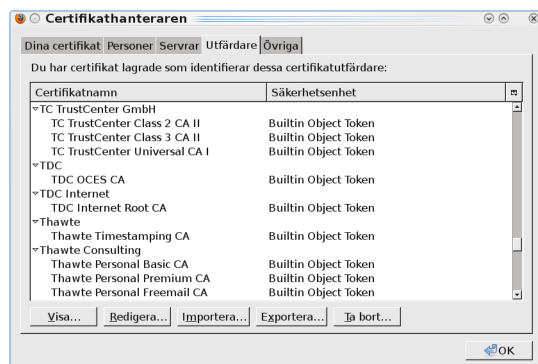
Metoden med att autentisera kod baserat på signaturer är grundat på förtroende. Administratören litar helt enkelt på att en organisation eller person gör rätt. Vemsomhelst kan dock skapa en modul med vilka namn somhelst så det behövs en mekanism för att avgöra vem som paketerat en given kod. Det görs genom att paketeraren signerar koden digitalt med standard-Java-verktyg. Administratören kan sedan ange i systemet att en vissa signerare är betrodd. Skillnaden mot standard-Java är att OSGi kräver att alla resursen i en modul är signerade.

### 2.10.1 Systemet med publika nycklar

I princip är det samma signatur-och kontrollförfarande som används för nerladdning av Java-applettar, ActiveX-komponenter och programpaket för Linux-system, liksom för äkthetskontroll av Webb och mail-servrar. Linux-systemen använder dock PGP-nycklar medan de flesta andra signaturer baserar sig på X509-certfikat. Principen är dock samma och går under namnet PKI (Public Key Infrastructure). Signeraren använde sin privata nyckel för att signera och veriferaren använde signerarens publika nyckel för att kontrollerar vem som signerat koden.

PKI är hierarkiskt, dvs man kan bestämma sig för att ha förtroende för individuella signerare eller att ha förtroende för alls som en viss part har förtroende för. M.a. om organisationen A har förtroende för B och C (t.ex. underavdelningar till A), så kan man säg att att man litar på alla som A litar på, dvs B och C och ev. andra parter i framtiden. De sker genom att A signerar de certifikat som B och C använder för att i sin tur signera kod. Webbläsare kommer med

en lista på s.k Rotcertifikat som listar organisationer som i sin tur utfärdar certifikat kommersiellt. Genom att köpa ett certifikat från en av desa leverantörer blir man automatiskt betrodd vad gäller ens identiftet när någon besöker ens webbsajt.




Figur 5: Några rotcertifikat i Firefox-webbläsaren

OSGi föjer samma princip. Om man hitta nån brist här är så är det att certifikat kan stjälas. I standarderna kring X509 finns definierat hur certifikatsutgivare kan sätta upp tjänster för att hantera revokeringslistor. Den som misstänker att ens privata nyckel blivit stulet kan då anmäla detta till certikatsutgivaren så att certifikatet inte länge är betrott. Certifikat har en sista giltighetsdag för att minska risken för att certifikat kommer på villovägar, men det ger ett ytterst begränsat skydd eftersom det stulna certifikatet är giltigt tills dess det går ut.

I [12], sekt 2.3 finns en detaljera beskrivning av hur moduler signeras och verifieras och hur förtroendekedjan för certifikat fungerar.

## 2.11 Implementationer

Det finns idag flera implementationer av OSGi version 3 och 4, t.ex. Apache Felix [21], Concierge[19], Equinox [20], Oscar [23] och Knoplerfish[3] som alla är tillgängliga som öppen källkod, men med varierande villkort för utnyttjande (licenser).

Felix, Equinox and Knopflerfish är idag aktiva och implementerar version 4. Concierge implementerar stora delar av version 3 och är intressant för att det är en väldigt liten implementation.

Equinox är den största i kod räknat och är optimierad för stora applikationer. Knopflerfish är den äldsta och kanske mest mogna implementationen [38].

Därutöver finns ett okänt antal in-house och kommersiella implementationer.

## 2.12 Program gjorda i andra språk än Java

En nackdel med OSGi är att det enbart är avsett för applikationer skrivna i Java. Det hindrar OSGi från att bli "Industristandarden" om inte Java och JVM också blir det. Java är dock mycket stort idag och finns i de flesta mobiltelefoner, SIM-kort (det lilla kort man lägger i mobilen som kopplar den till abonnemanget m.m.), BluRay-spelar och mycket annat. Java (och JVM-baserade språk) representerar ändå bara en liten del av den programvara som finns ute i världen.

Andra teknologier som DotNet (C# m.fl.), C/C++ och många, många andra teknologier har mycket begränsat stöd i OSGi.

Eftersom OSGi förutsätter att applikationen är skriven i Java (eller annat JVM-språk) så måste en modul skriven i något annat språk vara direkt anropbar från en Java-applikation. Det finns i princip två sätt detta kan göras på. Det ena är native-kod (JNI) [25], dvs kod kompilerad till ett dynamiskt bibliotek på plattformen (t.ex. shared library på unix-liknande plattformar eller dynamic link library eller DLL på Windows). OSGi har explicit stöd för JNI via `Bundle-NativeCode`-direktivet [12, sekt 3.9]. Det är dock förenat med vissa restriktioner som inte gäller för Java-kod. T.ex. så kan inte samma kod laddas från flera buntar och det går inte att direkt styra när sådan kod tas bort från systemet [12, sekt. 3.9.1]. En konsekvens är att resurser som hålls av native-koden hålls längre ä nödvändigt.

Den andra metoden är att göra ett fjärranrop av något slag till en nättjänst. [34] beskrivs hur detta kan göras i små inbyggda Java-system.

## 2.13 Kommunikation mellan moduler

Till skillnad från distribuerade miljöer såsom DCOM3.1.2, CORBA, RMI, DCE-RPE m.fl. är tjänsteregistret i OSGi en lokal konstruktion som fungerar inom en och samma virtuella Java-maskin. Det är inte p.g.a. lättja man gjort den avgränsningen, utan för att distribuerade miljöer i grunden är opålitliga [36] och gruppen bakom OSGi har inte velat bygga OSGi på en opålitlig grund.

### 2.13.1 Fjärrtjänster

OSGi version 4 har stöd för fjärrtjänster [13, kap 13], till stor del baserat på en metod beskriven i [33]. Vilka protokoll och begränsningar som gäller beror vilka protokoll som har stöd i den aktuella OSGi-installationen.

Det svåraste problemet med fjärrtjänster är att hantera det fall att kommunikationen inte fungerar [41]. OSGi-lösningen är att avregistrera en sådan tjänst. Eftersom det är ett scenario som varje välskriven OSGi-bunt bör hantera, skall detta i teorin vara transparent. Fjärrtjänster i OSGi är nytt och därför ganska oprövat i praktiken. Varningarna från [41], bör därför finnas i bakgrunden för den som tänker sig att bygga distribuerade system, vare sig de baseras på OSGi, eller något annat. Metoden med att avregistrera buntar vid kommunikationsproblem gör inte lösningen transparent, men den flyttar problemet till en annan och mera strukturerad abstraktionsnivå. En modul kan inte användas i fjärrsammanhang utan att den deklareras med den avsikten [13, sekt 13]

Beträffande kommunikation med fjärrtjänster stöder OSGi deklarationer av avsikter ("intents") [13, sekt 13.3]. Avsikterna ställer krav på ramverket beträffande olika aspekter av kommunikationen mellan modulerna, såsom t.ex. kryptering. I OASIS-standarden SCA (Service Component Architechture) ingår underspecifikationen SCA Policy Framework[28] som beskriver de standardavsikter som OSGi rekommenderar att implementationen av OSGi-ramverket noga bör följa.

Deklarationen i komponenten anger hur den skall publiceras. Det kan ske på flera sätt, t.ex. RMI eller som Web Service, för att ta några exempel [13, 15].. Detta i kombination med krav från den anropande komponenten bestämmer vilken ändpunkt som används. I [13, s14] finns en tabell med exempel på hur

tjänstekonfigurationer samtolkas. Motstridiga konfigurationer leder till att ingen förbindelse skapas.

## 2.14 Tjänstespåraren och deklarativa tjänster

Ett stort problem med att använda tjänster i OSGi är att det inte är säkert att de tjänster en modul behöver är tillgängliga när de begärs, t.ex. på grund av att tjänster initieras i en annan ordning än vad man förväntar sig. Det finns några sätt att hantera detta. Startnivåer är ett sätt men med ett stort antal moduler blir det snabbt komplicerat, så det använder man i regel för att se till att administrationsmoduler startas innan applikationsmoduler.

OSGi innehåller en tjänstespårare (service tracker) [13], sekt 701.3som gör det möjligt för en modul att prenumerera på information när en tjänst eller modul blir tillgänglig. Tyvärr kan [27], s87-93 konstatera att det krävs väldigt mycket och svårläst kod för att använda tjänstespåraren, så de rekommenderar att man använder deklararativa tjänster istället.

Med deklarativa tjänster [13, sekt 112], är det inte den modul som behöver en tjänst som anropar registret eller behöver vänta på att tjänsten blir tillgänglig. OSGi kommer att aktivera de moduler som behövs och när de är tillgängliga kommer den väntande modulen att bli aktiverad. Bara några få rader XML krävs för detta, som [27], s247-256 ger exempel på. Andra fördelar är att OSGi inte behöver installera moduler om de inte behövs och att uppstartstiden för systemet blir kortare, när färre moduler behöver laddas[13, sekt 112.1].

Eftersom det är ett koncist och enkelt format kan man tänka sig att använda dessa deklarationer för att analysera beroenden i ett OSGi-system grafisk eller på annat sätt, något som är mycket svårt när beroendena är skrivna i Java eller något att programmeringsspråk.

Ytterligare ett sätt att kontrollera beroenden finns, även om jag har haft svårt att hitta referenser utanför OSGi-specifikationen. Blueprint Container Specification [13], sekt 121 anger ett sätt att styra ihopkopplingen av komponenter helt utanför de berörda komponenterna. Kopplingen sker genom en i grunden enkel mekanism som går under namnet Dependency Injection (ung. beroendeinjisering), populariserad via bl.a. JEE-ramverket Spring, på vilken OSGi-specfikationen bygger. DI fungerar så att ramverket skaffar fram referenser till de obekt som behövs via deklarationer i XML-filer och sedan anropar get/set-metoder i de objekt som behöver referenserna. Även deklarativa tjänster bygger på DI.

## 2.15 Ramverks-API:et

Det finns ett antal sätt för program att arbeta med de OSGi-objekt som finns i ramverks-API:et. För den utvecklar OSGi-moduler bör dessa API:et undvikas för att inte göra sig beroende av OSGi i onödan och istället använda sig av t.ex. deklarativa tjänster för att koppla samma moduler.

## 2.16 Andra JVM-språk än Java

Det går att kompilera många andra språk än Java till JVM-kod och klassfiler, som t.ex. de populära eller klassiska språken Ruby, Python, COBOL, Lisp och Forth.

Vissa språk har utvecklats som komplement till Java, t.ex. Groovy eller enbart (till en början) bara för JVM. t.ex. Scala. I teorin borde dessa språk kunna användas i en OSGi-miljö, men det förutsätter att implementationerna inte ställer krav på sin körtidsmiljö som står i strid med OSGi. Seriösa försök att kunna köra JVM-implementationen JRuby under OSGi pågår [8]och det finns också exempel på hur man kan använda Scala i OSGi [2] och [2], liksom som författarna hävdar att dom framgångsrikt producerat OSGi-moduler i Groovy och Scala[10]. I exemplet använder man Groovy, som är ett språk som helt Java-kompatibelt, vilket förmodligen förenklare proceessen med att skapa en OSGi-modul. Slutsatsen man kan dra av dessa referenser är att användning av icke-Java-språk kan vara problematiskt, men att det kan komma att ändras till det bättre.

## 2.17 Problem och nackdelar med OSGi

OSGi är inte perfekt, trots att det bygger på mycket lång erfarenhet från organisationer som arbetar med komplexa programvaror, t.ex. Ericsson som är kända för att ha drivit modularisering långt redan på 1970-talet med AXE-växeln.

Vissa av nackdelarna är nackdelar sett från ett visst perspektiv. Det kan vara stora system, små system, hur kritiskt ett system är och så vidare. Om man kan ta ner ett system för kortare eller längre tid så behöver man inte funktioner för att byta ut program i körtid. En liten applikation kan byggas (kompileras) monolitiskt och där är OSGis stöd för att deklarativt hitta en implementation med rätt version överflödigt. En liten observation där är att många applikationer börjar i liten skala och växer och därmed växer de problem som måste lösas.

- [35] tycker att OSGi krånglar till det. Poängen är att OSGi kan vara för stort för en enkel applikation och att det går att modularisera program utan OSGi. Det är dock inte OSGi:s fel om man använder det fastän man inte behöver det.

- OSGi definierar hur versionsnummer skall se ut och begränsar det till tre delar (större, mindrem minsta och kvalificerare) och det passar inte alltid och framförallt inte för redan existerande kod. Ett exempel på det är OSGis tolkning att 1.0.0 $<$ 1.0.0.alpha medan en annan vanlig tolkning är att 1.0.0.alpha kommer före 1.1.0. Den senare konventionen används av bl.a. byggsystemet Maven [22]. Ett försök att ena dessa konventioner är att definiera hur man namnger versioner på ett sätt som fungerar med både Maven och OSGi. Det går under namnet OmniVersion [16]. Versionen “1.0.0” blir omöjlig med denna konvention eftersom dess tolkning är motstridig. Att det är just Maven och OSGi’s versioner man försöker ena beror på Maven sedan några år är det dominerande verktyget för att bygga Java-program inom den s.k. open-source-världen och då speciellt Apache-projektet som troligtvis är den största enskilda samlingen av öppen källkod inom Java-världen [18]. Samtidigt är Eclipse dominans beträffande grafiska utvecklingsverktyg mycket stort och Eclipse är numera en OSGi-produkt.

- Man måste ange metadata för alla bibliotek och då måste alla beroenden anges vilket kan vara väldigt många [38].

- Det är knepigt att anpassa bibliotek, som använder klassladdarna på ett avancerat sätt, till OSGi [38].
- Enbart för Java Virtual Machine, t.ex. använder OSGi den dynamiska klassladdningen som saknar motsvarighet i DotNet [38].
- Svårt att få att fungera med existerande applikationer. Problemet kan dock vara att applikationen är dåligt strukturerad och därför inte går att passa in i ett ramverk som kräver struktur.
- OSGi förutsätter lokala anropskonventioner, dvs att anropen sker inom samma JVM. Med andra ord är OSGi inte distribuerat förrän i den senaste versionen av standarden, som väl får betraktas som oprövad så här långt.
- Problem med att ladda moduler kan vara svåra att diagnostisera [12, sekt 3.12.3].

# 3 Jämförelse med andra modeller

Det finns likheter, men också olikheter med andra generella komponentramverk. Några av de mest kända av dessa är CORBA [37], JEE, Web Services och de flesta större operativsystem har mekanismer som i viss mån liknar det som finns i OSGi. Till dessa hör:

- Laddning av kod. Operativsystem stöder detta, inklusive dynamisk laddning och länkning av bibliotek.
- Åtkomstkontroll. De flesta operativsystem stöder detta, men hur det fungerar och vilka aspekter som hanteras varierar mycket. Det handlar om åtkomstkontroll för olika objekt inom operativsystemet såsom filer, processer, m.m. grundat vilka behörigheter som tilldelats en process eller tråd, vilka restriktioner som satts på enskilda objekt eller sökvägar, behörigheter som tilldelats viss kod.
  - Auktorisering. Under vilka förutsättningar kan vissa operationer utföras.
  - Autentisering. Å vems vägar utförs en viss operation. "Vem" kan vara en identifierad mänsklig användare, ett systemkonto, en maskin, en adress på ett nätverk.
  - Verifiering av kod. Det är idag vanligt att program begränsas av vem som gjort programmet. Digitala signaturer (ref) används för att på ett säkert sätt knyta en person eller organisation till en viss paketerad version av ett program.
- Administration av system. Hit hör installation av programkod, vem som får installera, vad som får installeras och ytterligare begränsningar som kontroll av beroenden mellan program.

## 3.1 Modulnivå

OSGi är inte enda ramverket för att stöda modularisering så en jämförelse med andra några populära ansatser kan vara på sin plats.

### 3.1.1 Java Enterprise Beans

Med Java följde jar-formatet för att paketera och distribuera kompilerad Java-kod. Det är dock bar en enkel behållare utan någon information om relationer till annan kod, versioner etc. Det som finns är signering av komponenter för att kunna garantera integritet och ursprung. Jar ingår i den variant om Java som kallas J2SE (Java 2 Standard Edition).

Därefter definierade Java 2 Enterprise Edition (J2EE, numer JEE) för att standardisera förpackning och distribution av komponenter i stora system. Jar-filer paketeras där i nya paketformer under namnet WAR (Web ARchive) och på ytterligare en nivå i EAR (Enterprice ARchive). Innehållet i en WAR är en WebbApplikation och en eller flera sådana kan levereras i en EAR. Med hjälp av klassladdningsmekanismen blir programmen i varje applikation isolerade från varandra, även om de har samma namn och en och samma webbserver kan således serva flera webbapplikationer utan att riskera konflikter på grund av globala (static-) variabler. Tjänster i J2EE registreras i en namntjänst (JNDI) via relativt komplexa beskrivningar i XML-filer som levereras tillsammans med applikationerna eller, i senare versioner av JEE, via deklarationer i Java-koden (Java-annotations). Att varje webbapplikation har en egen version av gemensam kod gör att mycket minne går till spillo och WAR-filerna är för stora för att fungera som en bra förpackningsenhet [27, xxi].

### 3.1.2 OLE, COM, ActiveX och DCOM

COM är Microsofts gamla men framförallt kommersiellt framgångsrika, komponentmodell. Det är ett antal konventioner och API:er för att definiera dynamiskt laddningsbara moduler som sedan kan användas i olika applikationer [42].

Föregångaren till COM är OLE (Object Linkage and Embedding), en teknik för att kunna bädda in program i program, t.ex. ett kalkylark i ett textdokument. OLE är komplext [42, s 27f]och mycket större vad som är lämpligt för en generell komponentmodell.

ActiveX är en uppstädning av OLE, bygger på COM och definierar de gränssnitt som behövs för att kunna ladda ner COM-komponenter i framförallt webbläsare. ActiveX är inte DCOM komponenterna bara laddas ner via nätverket, men exekveringen är normalt lokal. COM-komponenter registreras i Windows register ("registry") innan de kan användas. Därefter kan de instantieras. Beroende på hur komponenten är definierad kan detta ske lokalt i samma process, i en annan process på samma maskin eller på en annan maskin. En COM-komponents klass definieras av dess GUID (Global Unique Identification) som är ett långt bitmönster som konstrueras så att sannolikheten för att nån annan utvecklare skulle komma fram till samma id är försumbar.[42, s 12]

COM är dock idag ingen vanlig standard för dator-dator-gränssnitt, ens i Windowsdestor vanligare för lokala komponenter på Windows-plattformen. För kommunikation över nätverk driver Microsoft istället på Web Services som standard.

Istället för att gå igenom alla aspekter av COM så skall jag ta upp de största skillnaderna, Gränssnitt i COM är fasta och förändras inte. Därmed finns inte konventionen för kompatibla versioner i COM som finns i OSGi. Istället för att förändra gränssnitt skapar man nya gränssnitt om man till exempel vill lägga till en metod [42, s 69-70]. Man använder vidare inte arv för att definiera flera

aspekter av gränssnitt. Istället definierar man flera gränssnitt. [42, s 88]. Vid gränssnittsarv i Java så typomvandlar (eng: cast) man en referens för att få tillgång till en annan typ om objektet erbjuder mer än ett gränssnitt. I COM använder man ett generellt gränssnitt (IUnknown) som COM-gränssnitt implementerar. Där finns metoden QueryInterface för att fråga efter referenser till andra gränssnitt [42, s 81]. På låg nivå ser ett COM-gränssnitt ut som en abstrakt C++-klass med virtuella metoder [42, s75], men man använder typiskt ett speciellt verktyg (MIDL-kompilator) för att kompilera en gränssnittsdefinition [42, s 91ff].

COM använder referensräkning för att veta om ett objekt kan tas bort eller inte [42, s82f]. Instantiering av objekt sker via en funktion kallad Service Control Manager som kan skapa nya instanser lokalt eller till och med på andra datorer i nätverket förutsatt att den som begär tillgång till ett objekt har rätt behörigheter [42, s 235].

I stort sett alla programmeringsspråk som finns på Windowsplattformen kan använda COM-komponenter. Däremot är det inte lätt att skapa COM-servrar i alla språk.

Förutom den generella funktionen att kommunicera med tjänster på andra maskiner, definierar COM även stöd för autentisering och distribuerade transaktioner.

### 3.1.3 CORBA

Common Object Request Broker Architecture är en ambitiös standard för distribuerade system på alla typer av plattformar och språk. Till skillnad från COM som i praktiken bara är för Windows har CORBA bra stöd för de flesta operativsystem, men bara de största programmeringssråken verkar ha bra stöd för CORBA. Det har i stort sett varit begränsat till C, C++, Java och COBOL även om det finns udda inkompletta implementationer för andra språk också.

Ett antal grundtjänster finns för att registrera och komma åt CORBA-objekt. Naming Service är en tjänst för att registrera CORBA-objekt med namn och slå upp med namn. Dessutom finns tjänster definierade för det mesta, från händelsehantering till distribuerade transaktioner och autentisering av användare och även domänspecifika API:er.

Precis som med DCOM är ett av grundkoncepten att en tjänsts plats skall vara helt transparent för applikationsutvecklaren.

CORBA används än idag, men mest inom system som redan är implementerade.

### 3.1.4 Web Services

Den senaste flugan[1] inom distribuerad systemarkitektur går under namnet Web Services. Alla webbaserade protokoll är inte webb-services, utan termen avser specifikt användningen av det XML-baserade protokollet SOAP ovanpå HTTP. Uppdelningen mellan Windows och andra operativsystem med DCOM och CORBA ledde till att industrin började leta efter en lösning som alla kunde enas kring. Visserligen finns det ett antal bryggor som kopplar ihop DCOM med CORBA, med det har inte blivit någon universallösning.

[1] [41], s 3 noterar att nya tekniker för att bygga distribuerade system tas fram med jämna mellanrum

Web Services blev kompromissen, där till och med konkurrenter som Sun och Microsoft kunde enas. Vid tidigare standarder försökte man knyta fördelar till den egna plattformen genom leverantörsspecifika utökningar.

Plattformsspecifika utökningar verkar inte har blivit ett problem med WEB services, men samtidigt blir en ny form av protokoll mera vanligt. Det är en grupp protokoll som går under namnet REST (Representational State Transfer). Fokus i REST ligger på att skicka data med ett fåtal anrop. Formatet är också mera oreglerat och behöver alltså inte vara XML. JSON (JavaScript Object Notation) är ett populärt format som listar objekt-attribut-par i en mera kompakt. enklare och mera läsbar (för människor) notation är XML. Orsaken tilll att man frångår XML är att XML och i synnerhet SOAP är väldigt "pratigt", dvs själva protokollet tar mycket plats i jämförelse med det data man skickar. Det har också visat sig vara svårt att skriva små och effektiva tolkare för XML, vilket gör traditionella (dvs SOAP-) baserade protokoll impopulära på mobiltelefoner och andra mobila plattformar där minne eller strömförbrukning är begränsande faktorer.

[ref REST, JSON.]

#### 3.1.5 Dotnet

Dotnet (.Net i marknadsföring) är en virtuell arkitektur som i många avseenden liknar Javas virtuella maskin. Grunden är bytekoden MSIL (MicroSoft Intermediate Language) som tolkas av en virtuell maskin på liknande sätt som Java. Till skillnad från Java är Dotnet i praktiken än så längen bundet till Windows även om det finns en implementation för flera plattformar finns som öppen källkod i Mono-projektet[6]. Det finns implementationer för Unix-system, men de ligger efter i utvecklingen och saknar endel av de Windows-specifika API:er som gör att DotNet-program inte nödvändigtvis går att köra på annat än Windows utan att portas. Dotnet på Windows har t.ex. bra stöd för COM för att erbjuda en mjuk övergång för utvecklare på Windows-plattformen, men den delen saknas i Mono[4] liksom endel andra delar som ingår i Microsofts version för Windows[5].

De kompilerade programfilerna i DotNet kallas assemblies (sg assembly) och leveras med en namnsättning liknande den för Windows-program i allmänhet med suffix som .exe och .dll [40, s403-404]. En assembly motsvarar en jar-fil i Java i det att den kan innehålla kod och metadata motsvarande mer än en källkodsmodul. En assembly definiera också en namnrymd så att samma t.ex. klassnamn i olika assemlier innebär att de är olika klasser. En assemly har också, liksom paket och moduler i OSGi, en version, vilket gör att dotnet kan hantera flera versioner av samma kod samtidigt. Versionsnumreringen och filosofin bakom den liknar det som används i OSGi, men ramverket implementerar ingen tolkning liknande OSGi. Om en assembly refererar till en viss version så förväntas exakt den versionen finnas. Via s.k. policyfiler kan man ange att en viss version skall användas för att ersätta en annan. Det kan ske på flera nivåer, t.ex. i en applikation eller på systemnivå, genom att att administratören anger vilken version som ersätts och hur [7]. Dessutom är en assembly självbeskrivande så att körtidmiljön och verktyg kan se vilka beroenden och krav på omgivningen som ställs av denna [11].

## 3.2 Operativsystem

Uppstart och nedstängning av program hanteras lite olika i olika operativsystem. Drivrutiner för hårdvara är en mycket speciell klass som i regel hanteras helt inom operativsystemets kärna och liknar inget i OSGi. Program som en användare startar på sitt s.k. skrivbord är också utanför jämförelsen med OSGi. Däremot kan man göra jämförelser med tjänster.

OSGi hanterar allt inom en och samma Java-VM vilket är detsamma som en process i ett operativsystem. Tjänster på operativsystemnivå hanteras i regel som en tjänst per process.

### 3.2.1 Linux

De två vanligaste sätten att starta tjänster i Linux och andra Unix-liknande system är att man startar ett program som sedan ligger och lyssnar, eller att en generall tjänst startar en specifik tjänst när den behövs.

Tjänster som startar oberoende om de behövs listas i regel i /etc/init.d/rcN.d, där N är en siffra som står för en s.k. run-level, som symboliska länkar till program. När operativsystemet startas kommer dessa program att startas i en definierad ordning. Run-level anger hur mycket av operativsystemet och dess tjänster som skall vara uppe. De vanligaste nivåerna är single-user för systemunderhåll, non-interactive för servrar utan grafiskt användargränssnitt och interaktiv för arbetsstationer eller servrar med grafiskt användargränssnitt. Det går att byta run-level och operativsystemet kommer då att stänga och starta tjänster för att passa den valda konfigurationen.

Det andra sättet att starta tjänster, specifikt nätverkstjänster, är att använda inetd eller xinetd. Det är ett program som lyssnar på vissa nätverksportar och startar program när någon försöker använda tjänster. Det finns generellt stöd för att motarbeta överbelastning och begränsad åtkomstkontroll baserat på varifrån anropen kommer.

Som regel körs varje tjänst under ett eget användarid för att begränsa skadan vid intrång.

### 3.2.2 Windows

I många Windows-system startas tjänster genom att någon loggar in och startar ett program, men det finns bättre stöd än så via s.k, NT-services. Däri ingår att man definierar vilka program som skall startas och vilka beroenden som finns mellan tjänsterna. Om man begär att en tjänst skall startas och det är deklarerat att den tjänsten är beroende av någon annan tjänst kommer dessa andra tjänster att startas först. Det finns också stöd för att starta om tjänsten automatiskt om den skulle krascha. De flesta tjänster körs med alla privilegier, men det går att definiera att en tjänst skall köra med begränsade rättigheter genom att den startas med ett annat användarid än system-användaren NTSystem. Se även avsnittet om COM (3.1.2) och registring av COM-objekt.

# 4 Avslutning

OSGi är ett av många konkurrerarande ramverk. Att ramverket verkar etableras inom JEE (Java Enterprise Edition)-världen kan tolkas som att det kommer

att finnas kvar länge. Förutom just JEE finns nämligen inga andra avancerade etablerade standarder för hur återanvändbara komponenter paketeras inom Java och JEE har inte varit speciellt framgångsrikt som komponentstandard, till skillnad från framgångarna som utvecklingsplattform inom större företag. På källkodnivå finns Maven som konkurrerar om hur man gör komponenter tillgängliga, men det är endast beroenden och paketering som hanteras där och inte körtidsapspekterna, som är så viktiga för att kunna plocka ihop ett system från färdiga komponenter. Jar-formatet är framgångsrikt, men kanske mest för att det är standard och det är för fingranulärt för väldefinierade moduler.

Eftersom OSGi inte hanterar icke-Java-komponenter i någon nämnvärd utsträckning kommer det att finnas behov av andra paketerare på nivån ovanför OSGi, t.ex. Red Hat Package Manager, Windows Installer och operativsystembunda pakethanterare. I vissa miljöer är denna del redan avklarad av leverantören av t.ex. en robot och då kan de som utvecklar för den miljön enbart koncentrera sig på OSGi-paketeringen. Själva hårdvaran och paketeringen av operativsystemet sköts av någon annan.

I och med stödet för fjärrkomponenter är förmodligen problemet med att OSGi varit användbart för relativt isolerade system löst. Mekanismen är rätt ny så tiden får utvisa om det är för komplext, eller om det i praktiken blir ett attraktivt sätt att utbyta information med komponenter skrivna för andra plattformar än Java. Alternativet till fjärrtjänsterna är att man programmerar direkt mot API:er för nätverkskommunikation, med mindre transparens som följd.

Om Java tappar i attraktivitet så lär OSGi falla med Java. Några egentliga sådana tendenser saknas dock idag där det mesta kretsar kring Java [39] och DotNet [29, s3,5]. Dagen trend mot flerkärniga processorer kan ändra detta om om Java och Dotnet inte matchar förväntningarna, men det som talar mot ett alltför snabbt trendbrott där är att man ofta använder de nya multi-core-maskinerna för att virtualisera istället.

Arbete pågår med att förbättra modulstödet i standard-Java i JSR 294 i riktning mot bättre stöd för OSGi (om än indirekt) direkt i Java [24].

# Referenser


[1] Computer Language Benchmarks Game.
http://shootout.alioth.debian.org/. hämtad 2010-03-10.

[2] Creating a domain specific language for osgi. I: *Roman Roelofsen's blog*.
http://romanroe.blogspot.com/2008/03/dsl-for-osgi.html.

[3] Knopplerfish. http://www.knoplerfish.org. hämtad 2009-11-12.

[4] Mono project - com interop.
http://www.mono-project.com/COM_Interop. hämtad 2010-03-31.

[5] Mono project - Compatibility with .Net framework.
http://www.mono-project.com/Compatibility hämtad 2010-03-31.

[6] Mono-projektets hemsida. http://www.mono-project.com/Main_Page.
hämtad 2010-03-31.

[7] Redirecting assembly versions. I: *.NET Framework Developer's Guide.* http://msdn.microsoft.com/en-us/library/7wd6ex19%28VS.71%29.aspx. hämtad 2010-04-01.

[8] Using JRuby in OSGi. I: *Trialox.org.* http://wiki.trialox.org/confluence/display/DEV/Using+JRuby+in+OSGi. hämtad 2010-04-01.

[9] Ariane 5 - flight 501 failure. http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf, 1996. hämtad 2010-03-27.

[10] Osgi with scala, java, groovy, maven and pax. I: *Turmoil Driven Development (blog).* http://www.turmoildrivendevelopment.com/2009/05/osgi-with-scala-java-groovy-maven-and.html, 2009.

[11] .Net Framework Developer's Guide - Assemblies. http://msdn.microsoft.com/en-us/library/hk5f40cthämtad 2010-04-01.

[12] OSGi Alliance. Osgi service platform release 4 version 4.2 core specification, June 2009a.

[13] OSGi Alliance. Osgi service platform release 4 version 4.2 compendium specification, August 2009b.

[14] Jon Byous. Java technology: The early years. http://java.sun.com/features/1998/05/birthday.html, april 2003.

[15] James Gosling, Bill Joy, Guy Steele och Gilad Bracha. *The Java Language Specification, Third Edition.* Addison-Wesley Longman, Amsterdam, 3 utgåvan, June 2005.

[16] Thomas Hallgren Henrik Lindberg. Equinox/p2/omni version. http://wiki.eclipse.org/Equinox/p2/Omni_Version, 2009.

[17] Matthew Hertz och Emery D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. I: *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ss 313–326, New York, NY, USA, 2005. ACM.

[18] http://apache.org. Apache.org. http://apache.org.

[19] http://concierge.sourceforge.net. Concierge:. http://concierge.sourceforge.net.

[20] http://eclipse.org/equinox. Equinox. http://www.eclipse.org/equinox.

[21] http://felix.apache.org. Apache felix. http://felix.apache.org.

[22] http://maven.apache.org. Maven. http://maven.apache.org.

[23] http://oscar.objectweb.org. Oscar project home page. http://oscar.objectweb.org.

[24] jcp.org. Jsr 294 (draft): Improved modularity support in the javatm programming language. http://jcp.org/en/jsr/detail?id=294, 2007.

[25] Sheng Liang. Java native interface: Programmer's guide and specification. 2003.

[26] Tim Lindholm och Frank Yellin. Java virtual machine specification, 1999.

[27] Paul Archer Simon McAffer, Jeff VanderLei. *OSGi and Equinox - Creating Highly Modular Java Systems*. The Eclipse Series. Addison-Wesley, 2010.

[28] OASIS. Sca policy framework version 1.1. http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1-spec-cd02.pdf, 2009.

[29] Yefim V. Pezzini, Massimo Natis. Trends in platform middleware: Disruption is in sight. http://www.gartner.com/it/content/754400/754413/trends_in_platform_middleware.pdf, 2007.

[30] Lutz Prechelt. Plat_forms: Is there a single best web development technology? http://page.mi.fu-berlin.de/prechelt/Biblio/platforms07-cacm-2009.pdf.

[31] Lutz Prechelt. Technical opinion: comparing java vs. c/c++ efficiency differences to interpersonal differences. *Commun. ACM*, 42(10):109–112, 1999.

[32] Lutz Prechelt. An empirical comparison of seven programming languages. *Computer*, 33:23–29, 2000.

[33] Jan S. Rellermeyer, Gustavo Alonso och Timothy Roscoe. R-osgi: distributed applications through software modularization. I: *Middleware '07: Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, ss 1–20, New York, NY, USA, 2007. Springer-Verlag New York, Inc.

[34] Jan S. Rellermeyer, Duler Michael och Gustavo Alonso. Using non-java osgi services for mobile applications. 2008.

[35] Peter Rietzler. How to design software for flexibility, reusability and scalability without loosing kiss principles!, 02 2009R.

[36] Arnon Rotem-Gal-Oz. The fallacies of distributed computing explained.

[37] Jon Siegel. *CORBA 3 Fundamentals and Programming*. John Wiley & Sons, 2000.

[38] Ryan Slobojan. Neil bartlett on osgi, interview with neil bartlett, July 2009.

[39] TIOBE. Tiobe programming community index for march 2010, 2010.

[40] Andrew Troelsen. *C# and the .Net Platform*. 2003.

[41] Jim Waldo, Jim Waldo, Geoff Wyant, Geoff Wyant, Ann Wollrath, Ann Wollrath, Sam Kendall och Sam Kendall. A note on distributed computing. Teknisk rapport TR-94-29, IEEE Micro, 1994.

[42] Marshall Brain William Rubin. *Understanding DCOM.* Prenctice Hall, 1999.