Pluginarkitektur för Igor Engraver

Magnus Johansson

Institutionen för informationsteknologi
Avdelningen för datalogi
Uppsala universitet
Box 311
751 05
Uppsala

Detta arbete har utförts vid
NoteHeads Musical Expert Systems AB
Grevgatan 22
114 53 Stockholm

Abstract


handledare: Cons T. Åhs
examiner: Sven-olof Nyström

goddänt:
Innehåll

1 Inledning ................................................................. 4
   1.1 Allmänt om Igor Engraver ........................................ 4
   1.2 Syfte ........................................................................ 5
   1.3 Rapporten .................................................................. 6

2 Vad är en plugin? ...................................................... 7
   2.1 Mikrokärnemönstret (Bushman 1996) .......................... 7
   2.2 Vanliga exempel på pluginer ...................................... 9

3 Ett plugin-API i allmänhet .......................................... 10
   3.1 Ett plugin-API delar .................................................. 10
      3.1.1 Vårdprogrammets funktioner
            (kommunikation plugin till vårdprogram) ................... 10
      3.1.2 Pluginens funktioner
            (kommunikation vårdprogram -> plugin) ................. 10
      3.1.3 Datastrukturer .................................................. 11
      3.1.4 Administrativa funktioner .................................. 12
   3.2 Filformat .................................................................. 12
   3.3 Versionshantering och beroendena ......................... 14
   3.4 Övriga aspekter att ta hänsyn till ......................... 14

4 Igor Engraver ............................................................ 15
   4.1 Rex - ett ramverk för LISP-applikationer ............... 15
   4.2 Igor - en applikation som bygger på Rex ............. 16
   4.3 Ett musikstycke i Igor Engraver ............................ 16
      4.3.1 Mer om layouter och absolut notationsdata ... 16

5 Ett plugin-API för Igor Engraver ................................. 17
   5.1 Vårdprogrammets funktioner
       (Kommunikation Plugin till Igor Engraver) .............. 17
      5.1.1 Databasmodul ................................................. 18
      5.1.2 Användargränssnittsmodul ............................... 19
      5.1.3 Kommunikationsmodul .................................... 19
   5.2 Pluginens funktioner
       (Kommunikation Igor Engraver till plugin) ............... 19
   5.3 Datastrukturer ...................................................... 20
   5.4 Administrativa funktioner ...................................... 20

6 En NIFF-importsplugin ............................................... 21

7 Överblick över några pluginarkitekturer ..................... 22
   7.1 Historia ................................................................... 22
   7.2 Netscape Navigator 4.x ......................................... 22
   7.3 Adobe Photoshop .................................................. 23
   7.4 Finale 98 .............................................................. 24
      7.4.1 Allmänt ............................................................. 24
      7.4.2 Laddning ........................................................ 24
      7.4.3 Livscykel ......................................................... 24
      7.4.4 API ................................................................. 25
7.5 AutoCAD .............................................. 26
    7.5.1 Att skapa en ObjectARX-applikation. ............ 27
    7.5.2 Mer om databasen och filformatet. ............... 28
    7.5.3 Versionshantering ................................ 28

8 Slutsatser ............................................. 29
1 Inledning

Det här examensarbetet handlar om att utöka Igor Engraver med en plug-inarkitektur så att i slutändan tredjepartsleverantörer kan leverera utökningar till programmet. En plugin är en programvara som utökar ett befintligt program med ny funktionalitet, till exempel förmåga att hantera nya filformat. Igor Engraver är ett program för att skapa notskrift. De huvudsakliga konkurrenterna i denna kategori är Finale\textsuperscript{1} och Sibelius\textsuperscript{2}.

1.1 Allmänt om Igor Engraver

Igor Engraver är en produkt av Noteheads AB som grundades 1996 av Peter Bengtson. I egenskap av kompositör var han inte nöjd med de notskriftsprogram som fanns på marknaden, och i egenskap av programvaruutvecklare hade han möjligheten att göra något annat.


![Exempel på ett layout i Igor Engraver](image-url)

**Figur 1:** Här väljer du musiker och skapar layouter.


\textsuperscript{1}http://www.codamusic.com/coda/. (Access: 01-06-02)
\textsuperscript{2}http://www.sibelius.com. (Access: 01-06-02)
\textsuperscript{3}Vanligtvis fem parallelle linjer på och mellan vilka man skriver noterna.
Figur 2: Inmatning av noter.


1.2 Syfte

Syftet med detta examensarbete är att utreda och ge förslag till hur en pluggarkitektur för Igor Engraver kan komma att se ut.

1.3 Rapporten

Det här dokumentet börjar med att försöka reda ut vad en plugin är för något, samt relatera pluginer till existerande teori (Kapitel 2). Därefter kommer en mer detaljerad utredning kring de delar som utgör ett plugin-API (Kapitel 3). Denna del bygger på de upptäckter jag gjort när jag studerat ett antal befintliga pluginarkitekter. Sedan beskrivs Igor Engraver (Kapitel 4) för att läsaren ska få den information som är nödvändig för att förstå nästa del, ett förslag till pluginarkitektur för Igor Engraver (Kapitel 5). Även den exempelplugin jag skrivit beskrivs (Kapitel 6). Allra sist beskrivs de befintliga pluginarkitektererna jag kikat på (Kapitel 7).
2 Vad är en plugin?


Valença sätter också upp några krav som måste uppfyllas för att ett plugin-system ska vara "äkta":

1. Applikationen måste vara oberoende av pluginerna.
2. Pluginerna och applikationen måste betrakta varandra som "svarta lådor".
5. Applikationen bör definiera vilka tjänster som tillhandahålls till pluginen via gränssnittsdefinitioner.
6. En plugin måste i sig vara komplett, d.v.s. om pluginen inte i sig själv gör nytta utan kräver ytterligare funktionalitet för att vara användbar är det inte en "äkta" plugin.
7. En plugin bör skapa en ny plugininstans (ny tråd eller process) för att hantera varje uppdrag den får.

I praktiken implementerar man ofta en plugin som ett delat bibliotek (shared library), men man behöver inte göra det.

Några exempel på vad en plugin bör tillhandahålla är

1. information om vad den gör,
2. en funktion som ska anropas när pluginen behöver göra något,
3. ett sätt att hantera flera samtidiga invokationer av samma plugin (till exempel kan en webbsida innehålla två flashanimationer), och
4. en funktion för att initialisera pluginen.

2.1 Mikrokärnemönstret (Bushman 1996)

Jag har lyckats hitta ett designmönster (design pattern) som ganska bra beskriver pluginarkitekturer. Mönstret i fråga är mikrokärnemönstret (microkernel pattern). Alla exempel jag har sett på detta mönster har handlat om operativsystem, men mönstret kan lika bra användas för att beskriva applikationer med ett plugingränssnitt.

Mikrokärnemönstret är tillämpbart på programvarusystem som måste ha förmåga att anpassa sig till växande systemkrav. Det separerar en kärna med minimal funktionalitet från utökad funktionalitet och kundspecifika delar. Mikrokärnan fungerar också som en kontakt man kan plugga in sådana utökningar i,
och den koordinerar hur utökningarna samarbetar. Mikrokärnemönstret tillhandahåller en programvarumiljö som är “plug’n play”. Man kan lätt plugga in utökningar och integrera dem i programvarusystemet.

![Diagram of Mikrokärnemönstret](image)

Figur 3: Mikrokärnemönstret


Interna servrar utökar mikrokärnans funktionalitet. I ett operativsystem är bland annat drivrutiner interna servrar. Interna servrar kan laddas efter behov.


Klienten är i operativsystemsfallet en applikation som använder sig av POSIX-systemanrop, och i applikationsfallet är det en plugin.
2.2 Vanliga exempel på pluginer

Det de allra flesta tänker först på när de hör ordet plugin är nog webbläsarplug-iner. En typisk webbläsarplugin gör att läsaren kan visa nya typer av data (MIME-typer), till exempel pdf, flash, postscript, etc. Det normala här är att själva skapandet av datat inte sköts av någon plugin, utan av separata program, t.ex. Adobe Acrobat. Allt pluginen gör är att visa/spela upp datat.

En annan programtyp som många kommer i kontakt med och som i de allra flesta fall har pluginmöjligheter är bildbehandlingsprogram. Den typiska pluginen för ett bildbehandlingsprogram implementerar något slags filter för bilder.
3 Ett plugin-API i allmänhet

I detta avsnitt försöker jag reda ut vilka problem som kan komma att uppstå när man ska designa en pluginarkitektur, och vad man ska tänka på. Detta bygger på de pluginarkitekturer som presenteras i den sista delen.

3.1 Ett plugin-APIs delar

Det finns fyra delar av ett plugin-API: De funktioner som tillhandahålls av värdprogrammet, de funktioner som tillhandahålls av pluginen, de datastrukturer som tillhandahålls av värdprogrammet, samt de administrativa funktioner som används för att ladda pluginen, kolla dess beroenden med mera. För var och en av dessa delar finns det ett antal möjliga lösningar som alla har sina för- och nackdelar. Nedan diskuteras dessa.

3.1.1 Vårdprogrammets funktioner (kommunikation plugin till vårdprogram)


3.1.2 Pluginens funktioner (kommunikation vårdprogram -> plugin)

Här har jag sett tre varianter på lösningar, varav två är snarlika.

Fast uppsättning funktioner  Den första varianten är att pluginen måste implementera en specificerad uppsättning funktioner som vårdapplikationen sedan kan anropa eftersom den känner till namnet. Netscape gör på detta sätt. Även
i pluginarkitekturer som inte har löst det på detta sätt måste, i princip, åtminstone en funktion kännas vid namnet. Detta eftersom pluginen måste få möjlighet att initiera sig och registrera saker i vårdapplikationen4 (se nedan).

Återanropsfunktioner (callback) Den andra varianten är att pluginen har möjlighet att registrera återanropsfunktioner i vårdapplikationen. Det som varierar mellan pluginarkitekturer som har löst det på detta sätt är hur många ställen (fåspunkter) i vårdapplikationen som pluginen har möjlighet att lägga till återanropsfunktioner. Det allra enklaste, som till exempel Finale (Coda Music 1998) har löst det, är att man bara tillåter pluginen att lägga till nya menyval.


3.1.3 Datastrukturen


Det finns två viktiga frågor att ta ställning till när det gäller vad en plugin ska tillåtas göra med vårdprogrammets datastrukture.

4Detta är inte helt sant om applikationen är skriven i ett språk som tillåter att kod körs i samband med att den laddas, till exempel LISP.
1. Ska pluginer få lägga till nya datastrukturer till vårdprogrammet? Ett konkret exempel på när detta är tillåtet är AutoCAD som tillåter att en plugin lägger in nya typer av objekt i denna databas.

2. Ska pluginer få modifiera befintliga datastrukturer i vårdprogrammet? Ska en AutoCAD-plugin exempelvis få ändra representationen av en linje i databasen?

Svaret på den andra frågan (ska pluginer få modifiera datastrukturer?) är nej. Ett vårdprogram kan ha flera olika pluginer laddade samtidigt, och om dessa då tillåts göra förändringar av datastrukturer kommer det att antingen leda till att en plugin gör ändringar som är inkompatibla med en annan plugins ändringar, eller till att ändringarna är inkompatibla med vårdprogrammet.

Svaret på den första frågan styr till stor del hur kraftfulla pluginer man kan skriva. Kan pluginer inte lägga till egna datatyper till vårdapplikationen blir de begränsade att lägga till, ta bort, och modifiera data de de typer som redan finns i vårdprogrammet. I AutoCAD- exempel skulle det inte vara möjligt att lägga till nya typer av räkningsobjekt.

Fördelen med att inte tillåta pluginer att lägga till egna datatyper är att man slipper de problem som inträder när tillåtenhet för att använda ett plugin på objekt av en typ man inte kan benytta. Detta kan vara problematiskt, och detta gäller inte endast i vårdapplikationen, utan även i andra system som läser in DNA-data

1.3.4 Administrativa funktioner


3.2 Filformat

Om en plugin kan påverka vilka datatyper som kan lagras i vårdapplikationens databas leder det till nya frågeställningar:

5Om pluginen är skriven i ett språk som låter kod köra vid laddning behövs inte detta.
1. Vad händer om man försöker läsa in ett dokument där pluginspecific information finns lagrad och pluginen som genererade denna information inte är laddad?


Jag har stött på tre svar på den första frågan.

1. Det enkla sättet att hantera detta är att helt enkelt vägra ladda filen. Detta är enkelt att implementera, men inte så kul för användaren som då inte ens kan se den information i filen som denna information faktiskt kan hantera.

2. En annan variant är att bara ladda de delar applikationen kan hantera och kasta bort resten. Självlklart måste användaren få information om att detta sker. Detta är också ganska enkelt att implementera, men kan leda till informationsförhust om användaren sedan sparar filen.


Svaret på fråga två borde vara självklart ur användarsynvinkel. Användaren måste kunna få reda på vad han/hon saknar när en fil laddas.

Även versionsnumret på pluginen som skapade informationen bör vara tillgängligt för värddapplikationen. Även detta för att användaren ska få reda på vad som saknas. Om versionsnumret för den plugin som är laddad är nyare än den version som skapade informationen i filen kan användaren få möjlighet att försöka ladda filen med hjälp av den nyare pluginen.
3.3 Versionshantering och beroenden

En plugin kommer med största sannolikhet att ställa krav på vilken version av värdprogrammet den behöver. Detta leder till att värdprogrammet bör ha något sätt att kontrollera en plugins beroenden innan det laddar den, alternativt att använda för detta läggs över på plugierna själva, det vill säga att de avslutar sig själva om de upptäcker att något beroende inte är uppfyllt. Problemets komplicerade ytterligare om man vill tillåta att en plugin själv kan vara en pluginvård, det vill säga man kan ha pluginer till pluginer. Detta är tänkbara om man har en stor plugin som utökar värdprogrammet med mycket ny funktionalitet.

Ett exempel på detta skulle kunna vara en plugin som utökar ett bildbehandlingsprogram med en ny abstraktion för färger. Man kan då vilja skriva nya pluginer till denna som opererar på den nya fargabstractionen. En sådan stor plugin kan tänkas utöka det API som tillhandahålls av värdapplikationen och ge andra pluginer nya funktioner att anropa. Allt detta ställer krav på att man inte bara letar efter pluginer som förblir beroende av det API som det nya pluginet bygger på.

Ett sätt att lösa detta problemet är att man låter plugierna själv avläsa vilken version av plugin som de avhänger av och avläsa den avläsning i att de inte är ett problem. Detta gör det lite svårare för användaren att hantera pluginer i rätt ordning eftersom värdprogrammet inte gör det.

Man kan dock också enkelt lösa detta problem genom att låta pluginen kontrollera vilken version av plugin som är tillgänglig. Detta kräver dock att man kan komma åt pluginens binärer och att pluginens autonoma funktioner är tillgängliga.

3.4 Övriga aspekter att ta hänsyn till


Om man låter en plugin legga till egna menyer måste man tänka på att man inte skall ta det. Skapa pluginen får inte legga till menyer som om det inte skulle behöva en plugin. Detta kan också vara en problem som kan uppstå vid utvecklingen av en plugin.

Om man låter en plugin legga till egna menyer måste man tänka på att man inte skall ta det. Skapa pluginen får inte legga till menyer som om det inte skulle behöva en plugin. Detta kan också vara en problem som kan uppstå vid utvecklingen av en plugin.

---

6Ett sätt att skriva mer musik för stränginstrument.
4 Igor Engraver

Igor Engraver är internt uppdelat i ett antal moduler. De viktigaste modulerna är:

1. Igor. Det är denna modul som samlar ihop alla de andra och gör en applikation av det.


De olika modulerna kommer i varierande grad att beskrivas här nedan. Jag väljer att börja med Rex, eftersom nästan allt annat är skrivet med hjälp av de mekanismer som Rex tillhandahåller.

4.1 Rex - ett ramverk för LISP-applikationer

Rex kan ses som Igor's mikrokärna. Ovanpå Rex kan man i princip bygga vilken dokumentbaserad applikation som helst. Rex tillhandahåller funktioner för undo/redo, hantering av mus och tangenter, klipp och klistra, notshantering ( använts till exempel för att ett fönster ska få veta att det är dags att uppdatera sig), tillståndshantering (en applikation kan ofta befinna sig i ett av flera tillstånd, till exempel inmatningsstillstånd), med mera. Även funktioner för specifikt finns här. Specifikt används när, i princip, ett och samma objekt ska finnas representerat på flera sätt. Till exempel i Igor är noter i en layout (se nedan) en specifik av noter i det globala notationsdator. Detta gör att samma not kan finnas med i olika layouter och ha olika attribut i dessa layouter, men ändå vara synkroniserade så att när man ändrar notboken i ena layouten ändras den i alla andra också. Rex tar hand om all detta synkronisering.
4.2 Igor - en applikation som bygger på Rex


4.3 Ett musikstycke i Igor Engraver


4.3.1 Mer om layouter och absolut notationsdata

En layout skapas genom att man "speglar in" ett antal musiker i layouten. Dessa musiker innehåller notationsdata (eller så kan de vara tomma om man inte har lagt in något än). Detta notationsdata får ytterligare egenskaper i layouten. Till exempel kan man i layouten sätta färger på en not, vilket man inte kan göra direkt i den globala datavektorn som finns i musikern. Man kan se det som att notationsdata kan ha olika lokala attribut i varje layout, fast det i själva verket är samma notationsdata.

Den notationsinformation som lagras i den globala datavektorn är absolut i den mening att tonhöjder inte är representerade relativt några notlinjer, utan snarare som avståndet till den lägsta representerbara tonhöjden. Detta gör att samma notationsdata kan se olika ut i olika layouter, där hänsyn tas till tonart och klav för att avgöra var på notlinjen en viss not ska ligga. Notationsdata i en layout presenteras relativt en klav och en tonart.
5 Ett plugin-API för Igor Engraver

Jag väljer här att beskriva ett plugingränssnitt som inte tillåter pluginer att
lägga till nya datastrukturer. Detta på grund av de problem som diskuterats
tidigare.

Ett sätt att ta reda på vad ett plugin-API behöver bestå av är att skriva
en liten testplugin. Jag har skrivit en liten plugin till Igor för att importera ett
nytt filformat (NIFF, Notation Interchange File Format). Anledningen till att
det blev just en filimportsplugin är att det var vad Noteheads AB önskade sig.
Arbetet med denna plugin beskrivs i avsnitt 6. Den följande beskrivningen av
plugin-API:et blir därför i huvudsak utgående från min plugin.

Observera att den plugin jag skrivit inte är någon åkta plugin, eftersom den
just nu inte använder sig av något specificerat plugin-API, utan använder sig av
interna Igor-funktioner. Dock upplevs den som en åkta plugin eftersom den kan
laddas dynamiskt efter det att Igor har startat (detta beror som tidigare nämnts
på att Igor och pluginen är skrivna i LISP). I mikrokärne-terminer så pratar min
plugin (klanten) direkt med Igor och Rex (den externa och interna servern) och
inte via ett plugin-API (en adapter).

5.1 Världprogrammets funktioner
(Kommunikation Plugin till Igor Engraver)

Som jag nämnd i ett tidigare avsnitt är inte världsprgrammets funktioner särskilt
spännande. Det handlar mest om att besluta om vad man vill att pluginer ska
kunna göra.

Ett filimports-API behöver i första hand tillgång till Igors databas, samt
de datatyper som lagras i den. Detta behöver min plugin kunna göra för att
fungera:

1. Skapa stycke, musiker, instrument, notsystem, layout, ackord, tonhöjd,
   not, pans, artikulationer, bindebågar, legato, tupletar7, med mera.
2. Plocka ut notskriftsdata ur en musiker.
5. Ändra antal takter i ett stycke.
6. Ändra ett antal parametrar för en tupel.
7. Speglar musiker till en layout.
8. Bygga en layout. Detta innebär att Igor tar allt notationsdata och skapar
   en vacker visuell representation av det.

7Observera att jag inte menar tupletar i ett datavetenskapligt sammanhang, utan i musik-
sammanhang, det vill säga i huvudsak trioler. En triol är en grupp av tre noter som uppträ
samma taktdeel som två eller fyra vanliga av samma slag. Dock behöver det inte alltid vara
just tre noter, därför valde jag samma tuplet.
10. Ändra färg på ett visuellt objekt.
11. Fästa bilhang (attachments) till objekt.
12. Lägga till ett menyval.
13. Låta användaren välja fil att importera.
15. Skapa en logg.

Som man ser av listan ovan är det främst skapa, ändra och plocka ut saker som behövs. Jag väljer att dela upp plugin-API:et i olika moduler för att få överblick. Min modul kommer att behöva dessa moduler:

1. Databas-modul
2. Användargränssnittsmodul
3. Kommunikationsmodul (modul för att registrera återanropsfunktioner)

Naturligtvis kommer det på sig att behövas fler moduler än dessa, men inte för min plugin. Till exempel kommer en plugin som vill spela upp något med MIDI att behöva tillgång till en MIDI-modul.

Jag tänker inte göra någon uttömmande utredning kring vilka funktioner som bör finnas med i respektive modul, utan jag fokuserar på det som behövs för min filimportplugin. Jag tror databasmodulen kommer att vara ganska komplett när de delar jag behöver i min plugin är implementerade, medan layoutmodulen och användargränssnittsmodulen antagligen kommer att behöva ytterligare delar.

Samtliga API-funktioner bör implementeras som wrappers kring de funktioner som faktiskt används. Detta på grund av att interna ändringar i Igor inte ska påverka plugin-API:et.

### 5.1.1 Databasmodul


Min plugin har inget behov av att radera saker i databasen, men även sådana funktioner bör finnas här.

Det finns även behov av att komma åt speglar av objekt. I min plugin behöver jag till exempel komma åt en visuell representation av en paus, utgående ifrån den paus som finns i den globala datavektorn.

Det måste finnas ett sätt att fästa bilhang till objekt, till exempel bindebågar.

Eftersom detta kommer att bli en ganska stor modul kan det vara klö att dela upp den i mindre. Man kan till exempel tänka sig att bryta ut funktioner
för att skapa, ändra och radera enskilda notationselement till en egen modul, medan funktioner som opererar på större delar, till exempel takter, system och stycken, får ligga kvar.

5.1.2 Användargränssnittsmodul


5.1.3 Kommunikationsmodul


5.2 Pluginens funktioner

(Kommunikation Igor Engraver till plugin)

I min filimportspplugin är den enda funktion som behöver anropas av Igor den som kör igång importen. Allt annat sköts av pluginen själv. Därför säger inte min plugin särskilt mycket om hur kommunikationen från Igor till pluginen bör gå till. Det följande blir därför mest resonerande.

princip bara hålla reda på vilka pluginer som är laddade, medan för återanrops-
funktioner måste Igor ha en lista med funktioner för varje fästpunkt.

Den enda fästpunkt min plugin behöver är det menyval den lägger till i Igor.
Det snidigaste är att låta den funktion som lägger till menyvalet ta återanrops-
funktionen som argument.

5.3 Datastrukturer

Eftersom man i den första versionen av plugin-API:et inte kommer att kunna
lägga till egna datatyper i databasen blir denna del inte så mycket att orda om.
För LISP-API:et handlar det i princip bara om att dokumentera vilka data-
strukturer som pluginen ska ha tillgång till. All åtkomst till dessa datastrukturer
sköts sedan med hjälp av de funktioner som beskrivs i avsnitt 5.1.

5.4 Administrativa funktioner

Det enklaste sättet för Igor att få reda på vilka pluginer som finns tillgängliga är
att kräva att alla pluginer ska läggas i en speciell katalog. Vid start eller behov
kan Igor då ladda dessa pluginer.

Det som är snyggast vad det gäller versionshantering är att Igor själv kan få
reda på en plugins beroenden på något sätt för att sedan kunna ladda pluginerna
i rätt ordning om det finns beroenden mellan olika pluginer. Mitt förslag blir
därför att varje plugin måste implementera en funktion som talar om vilka
beroenden den har. Detta inkluderar såväl som vilken version av Igor den kräver,
som vilka andra pluginer och deras versioner den är beroende av. Givet denna
information kan Igor sedan avgöra om pluginen kan användas, eller om andra
pluginer ska laddas först.

Pluginen måste även ges möjlighet att initialisera sig. Tack vare att Igor är
skrivet i LISP så kan pluginerna köra kod i samband med att de laddas, men jag
förordrar trots det att pluginerna implementerar en speciell initialiseringsfunk-
tion som Igor anropar. Detta för att det då blir större likhet med ett framtida
C-API, som ju inte har möjligheten att automatiskt köra kod vid laddning.
6 En NIFF-importsplugin


I princip handlar problemet med att importera främmande filformat om att översätta från en abstraktion till en annan. Lyckligvis stämmer NIFF-formatet och Iogors interna format överens till ganska stor del, så i de flesta fall har det bara varit att göra en översättning från en NIFF-struktur till en direkt motsvarande struktur i Igor, medan i andra fall har jag varit tvungen att trixa en del. Ett sådant trix är till exempel hur matchningen mellan NIFF-instrument och Igor-instrument går till. Ett instrument i NIFF kan heta vad som helst, till exempel ”Part I”, ”RH1”, ”LH1”. Det finns inget lätt sätt att automatiskt hitta en motsvarighet i Igor i dessa fall, men oftast har användaren kunskap om vad stycket innehåller för instrument och kan därför själv matcha instrument via ett användargränssnitt. (av RH1 och LH1 kan man till exempel dra slutsatsen att dessa ”instrument” i själva verket motsvarar vänster respektive högerhanden på ett piano). Speciellt notscanningsprogram är (av naturliga orsaker) dälaughter på att ge en bra beskrivning av vad det är för slags instrument.


Jag hittade även ett antal buggar i NIFF-filer skapade av vissa program. Det tog tid att inse att det faktiskt var buggar i mina exempelfiler, och inte buggar i min importplugin.
7 Överblick över några pluginarkitekturer

I det här avsnittet ger jag en snabb överblick över några olika pluginarkitekturer och visar på hur de har löst problemen som tidigare beskrivits.

Huruvida de applikationer jag beskriver här passar in på mikrokärnmönstret vet jag inte eftersom jag inte har tillgång till dokumentation som beskriver dessa applikationer internt, men klart är att de delar som har med pluginarkitekturen att göra stämmer in på motsvarande delar i mikrokärnmönstret.

7.1 Historia


Företaget Silicon Beach med programmen Digital Darkroom och SuperPaint gjorde det möjligt att låta pluginerna ligga i egna filer, vilket gör det möjligt för pluginerna att ligga vart som helst, inte bara i systemmappen. Silicon Beachs plugindesign hade också versionnumrering, vilket gjorde det lättare att utöka APIet med ny funktionalitet.

7.2 Netscape Navigator 4.x

Netscape är ett välbelant program för de flesta. Det är precis som Internet Explorer och Opera en webbläsare.

Netscrapes pluginguide listar vad en plugin kan göra:

1. registrera en eller flera MIME-typer,
2. rita i en del av Netscapeönstret,
3. ta emot tangentertryckningar och mushändelser,
4. läsa data från nätverket via URLer,
5. skicka data till URLer (via POST),
6. lägga till hyperlänkar och "hotspots" som länkar till nya URLer, samt
7. rita i delar av ett HTML-dokument.


1. Leta efter en plugin med matching MIME-typ.
2. Ladda in pluginen i minnet.
3. Initiera pluginen.


När man sedan lämnar sidan så raderas instansen av pluginen. Om det var den sista instansen så tas även pluginen bort.

Netscapes API är ganska litet. Det består av 16 funktioner som pluginen måste implementera, samt 22 stycken funktioner som Netscape tillhandahåller.


7.3 Adobe Photoshop


Plugin-API-moduler för Photoshop kan delas in i nio kategorier:


Color Picker Dessa moduler kan lägga till ytterligare färgväljare (color picker) utöver de som tillhandahålls av Photoshop.

Import Importmoduler läser in bilder till ett nytt fönster i Photoshop. Dessa moduler kan användas till exempel för att få in en bild direkt ifrån en skärm, läsa in filformat som inte stöds direkt av Photoshop, eller kanske algoritmiskt generera en bild.

Export Exportmoduler används exempelvis till att skriva ut bilder på skrivare som inte stöds direkt, eller att spara en bild i ett nytt filformat.

Extension Extensionsmoduler kör en gång vid programs start och en gång när det avslutas och har oftast inte något användargränssnitt. Dessa används till exempel till att initiera någon möjlig (device). APIet för dessa är inte tillgängligt, utan används endast intern i Adobe.

Filter En filtermodul modifierar en utvald del av bilden. Detta är den vanligaste typen av pluginer.

9http://www.macromedia.com/software/flash/ (acc. 2001-11-02)
**Format** Dessa moduler ger photoshop möjlighet att spara i nya filformat. Dessa moduler skiljer sig från Import/Export-moduler genom att man kommer åt formatmodulerna vi menyval som Open/Save, medan Import/Export-modulerna hamnar under Import/Export menyarna.


**Selection** Selection-moduler bestämmer vilka pixlar som väljs i en bild.

Sammanfattningsvis kan man dela in photoshoppluginer i fyra kategorier: filter, färgväljare, pixelväljare, samt stöd för nya filformat.

### 7.4 Finale 98

Finale är tillsammans med Sibelius\(^1\) huvudkonkurrent till Igor Engraver. Det är det mest spridda notskriftsprogrammet på marknaden. Se Codas hemsida\(^2\) för mer information.

Den här beskrivningen av Finales pluginer gäller Finale 98 eftersom Finale 2000 PDK (Plugin Development Kit) inte finns tillgänglig i skrivande stund.

#### 7.4.1 Allmänt

En Finale plugin implementeras som en DLL-fil under Windows och som delade bibliotek under Mac/PPC. Ett antal av Finales verktyg är tillgängliga för pluginen, liksom nästan all notationsdata. En begränsning som Finalepluginer har är att de inte kan ge Finale features det inte redan har, till exempel "French beaming" och editering av fonttecken. I och med detta verkar det som att de heller inte kan göra saker som kan påverka filformatet. Pluginerna presenteras i Finale som menyal i menyn "Plugin".

#### 7.4.2 Laddning


#### 7.4.3 Livscykel

1. Pluginen laddas (se ovan).
2. Pluginens FinaleExtensionVersion() anropas för att kontrollera att pluginen var skapad med en PDK som stöds av denna version av Finale.
3. Pluginens FinaleExtensionInit() anropas så att pluginen kan initiera sig själv.

---

\(^1\)http://www.sibelius.com/
\(^2\)http://www.codamusic.com/coda/
4. Plugindens FinaleExtensionEnumerate() anropas så att Finale får reda på vilka menyval pluginen vill lägga till.

5. Initieringen är nu klar.


7.4.4 API


De funktioner som Finale tillhandahåller till pluginen är uppdelade i några olika kategorier:

EXTGIF oriented Denna del består av fyra stycken funktioner som verkar operera på enskilda notations-element.

EREGION oriented (Enigma Region) Tre stycken funktioner (RebeamRegion, RebarRegion, och GetCurERegions) som opererar på större notationsområden.

Database oriented Här finns sex stycken funktioner som används för att komma åt notationsdata (EData).

Dialog Related Här finns funktionalitet för att pluginmakaren ska kunna bygga sina egna dialogruttor. Denna del består av 14 stycken funktioner.

MIDI and Playback I denna del finns funktionalitet för att spela upp och modifiera MIDI-data. Den består av åtta stycken funktioner.

Progress bar Här finns tre stycken funktioner för att visa en förloppsindikator (progress bar).


Finale dialog boxes Ytterligare funktioner för dialogruttor, men till skillnad från de dialogrelaterade funktionerna så ger dessa funktioner bara tillgång till dialogruttor som Finale redan har definierat. Här finns sju stycken funktioner.

Miscellaneous Här finns tolv stycken funktioner som inte passar in i någon av de andra kategorierna. Bland annat finns här en funktion för att skrolla till en viss takt, samt funktioner för att uppdatera skärmbilderna.

Utöver dessa ca 60 stycken funktioner finns det dessutom i storleksordningen 140 st datatyper som en plugin kan behöva använda.
7.5 AutoCAD

AutoCAD är det mest spridda CAD-programmet på marknaden. Även om man kan använda AutoCAD som det är, så är det mycket vanligt att man snarare jobbar i någon plugin till AutoCAD. AutoCAD tillhandahåller i grunden de vanliga funktionerna för att skapa ritningar. Pluginer utökar sedan AutoCAD till någon speciell problemdomän, till exempel elektronik, vägar eller broar. Dessa pluginer säljs ofta som separata applikationer även om de har AutoCAD i grunden. Se Autodisks hemsida\footnote{http://www3.autodesk.com/adsk/section/0,616714-123112,00.html} för mer information.


AutoCAD hade möjlighet till pluginer redan innan termen plugin hade etablerats. Därför kallas pluginerna i ObjectARX-fallet helt eukelt för ObjectARX-applikationer.

ObjectARX tillhandahåller ett objektorienterat API som kan användas för några olika saker:

1. Man kan komma åt AutoCAD:s databas.
   En ritning i AutoCAD är en samling objekt som lagras i en databas. Dessa objekt behöver inte ha grafisk representation. ObjectARX har stöd för att komma åt dessa objekt och dessutom kan man skapa nya databasobjekt.

2. Man kan interagera med AutoCAD:s editor.
   Man kan registrera nya kommandon som då kommer att behandlas som om de vore AutoCAD:s egna. Dessutom kan pluginen reagera på en mängd händelser som inträffar i AutoCAD.

3. Man kan skapa användargränssnitt med Microsoft Foundation Classes (MFC).
   Pluginen kan dela MFC-biblioteket med AutoCAD.

4. Man kan använda gränssnittet AutoCAD har för multipla dokument (Multiple Document Interface, MDI).

5. Man kan skapa egna klasser.
   Det vill säga man kan under runtime lägga till nya klasser.

6. Man kan skapa komplexa pluginer.
   Genom så kallad "protocol extension" kan man utöka funktionaliteten hos existerande klasser.

7. Till sist kan man också interagera med andra programmeringsmiljöer. Till exempel Visual LISP och COM.

ObjectARX består precis som de andra arkitekturerna av en del som tillhandahålls av AutoCAD och en del som ska tillhandahållas av pluginen. ObjectARX-objekt kan klassificeras enligt följande:
1. AcRx
   Denna del innehåller klasser för uppstart och initiering, samt för att under-
körning kunna registrera och identifiera nya klasser. Även funktionalitet
för "protocol extension" finns här.

2. AcEd
   Denna del innehåller klasser för att definiera och registrera nya AutoCAD
kommandon. De nya kommandon man registrerar lagras i samma struk-
tur som AutoCADs inbyggda kommandon. Här finns också en klass som
övervakar editorn och talar om för pluginen när vissa händelser inträffar,
som till exempel ett kommando startar, slutar, eller avbryts.

3. AcDb
   Här finns databasklasserna. Det som lagras i databasen är till exempel
de grafiska objekt som utgör en ritning, men även objekt utan grafisk
representation men som ändå är en del av ritningen, till exempel lager,
linjetyper och textstilar. Databasen består av tre delar: Nio stycken sym-
boltabeller, uppslagsdelar (dictionary), samt en uppsättning variabler (200
st) som kontrolleras av AutoCAD. Både symboltabeller och uppslagsdelar
är en mappning mellan symbolnamn (textsträng) och ett databas- objekt.
Varje symboltabell innehåller objekt av en viss typ, till exempel finns alla
lager i en symboltabell. Uppslagsdelar är mer generella än symboltabeller
eftersom de kan innehålla objekt av vilken typ som helst. I varje databas
finns det alltid en uppslagsdel som kallas "Named Object Dictionary"
som kan ses som en innehålls- förteckning över hela databasen. AutoCAD
tillåter också att man har flera databaser laddade samtidigt.

4. AcGi
   Denna del innehåller klasser för att rita AutoCAD-objekt.

5. AcGe
   Denna del tillhandahåller klasser för matrisalgebra, samt grundläggande
geometriskt objekt som till exempel punkter, kurvor och ytor. Biblioteket
är indelat i två delar: en del för två dimensioner och en för tre.

En skillnad gentemot de andra pluginarkitekturerna är att den del som tillhan-
dahålls av pluginen bara består av en funktion. Pluginen lyssnar på meddelanden
från AutoCAD via denna funktion. Det finns ett antal sådana meddelanden,
till exempel meddelanden från editorn, från databasen, eller från ett objekt i
databasen.

7.5.1 Att skapa en ObjectARX-applikation.
Följande steg behöver tas för att skapa en ObjectARX-applikation:

1. Skapa klasser som implementerar nya kommandon.
2. Bestäm vilka AutoCAD-meddelanden som applikationen ska reagera på.
3. Implementera en funktion som AutoCAD kan använda för att prata med applicationen (acxEntryPoint()). Denna funktion motsvarar main() i ett vanligt C++-program. acxEntryPoint() dirigerar inkommande meddelanden vidare till rätt funktioner.

4. Implementera initiering.

5. Implementera avslutning.

7.5.2 Mer om databasen och filformatet.

Till skillnad från de pluginarkitekturer som beskrivits ovan så kan pluginönspecifika information lagras i AutoCADs filer. Det finns två filformat som AutoCAD använder: DXF (Data eXchange Format) och DWG (DraWinG). DWG är AutoCADs eget filformat, medan DXF är ett ASCII-baserat, öppet filformat. I båda fallen lagras hela AutoCADs "databas" i filen.


Det framgår inte av dokumentationen hur objekten taggas i filerna, men någon form av taggning måste förekomma för att AutoCAD ska kunna veta vilken typ av objekt som ska läsas in.


7.5.3 Versionshantering

8 Slutsatser

Under arbetet med detta examensarbete har jag kommit fram till att ett plugin-API kan sägas bestå av fyra delar. Det finns frågor associerade med dessa delar som måste besvaras när man designar ett plugin-API. Nedan listas delarna och de viktigaste frågorna som är associerade med varje del.

1. Funktioner som tillhandahålls av vårdapplikationen (hur pluginen kommunicerar till vårdapplikationen).
   (a) Vilka funktioner ska pluginer få tillgång till?

2. Funktioner som tillhandahålls av pluginen (hur vårdapplikationen kommunicerar till pluginen).
   (a) Hur ska kommunikationen gå till? Ska man ha en fast uppsättning funktioner som pluginen måste implementera? Eller ska man låta pluginen registrera återanropsfunktioner? Eller ska vårdapplikationen få skicka meddelanden till pluginen?

3. De datastrukturer som används.
   (a) Ska pluginer få lägga till nya datastrukturer?
   (b) Ska pluginer få modifiera befintliga datastrukturer?

4. Administrativa funktioner för laddning av pluginen, med mera.
   (a) Ska pluginer få bero på något annat än vårdapplikationen?
   (b) Vem är ansvarig för att se till att pluginens eventuella beroenden är uppfyllda? Pluginen eller vårdapplikationen?


De flesta pluginarkitekturer på marknaden är av enklare typ, men det finns också mer komplicerade, till exempel AutoCAD.
Referenser

Adobe (2000). *Photoshop SDK*
http://partners.adobe.com/asn/developer/gapsdk/PhotoshopSDK.html (Acc. 01-09-28).

Autodesk (2000). *ObjectARX for AutoCAD 2000i*
http://www3.autodesk.com/adsk/item/0,785550-123112,00.html (Acc. 01-09-28).

F. Bushman & R. Meunier et. al. (1996) *A system of patterns*

Coda Music (1998). *Finale 98 PDK*

Hood, Earl (1998). *Multipurpose Internet Mail Extensions*

Mounce, Steve (1997). *NIFF Page*
http://www.student.brad.ac.uk/srmounce/niff.html (Acc. 01-09-30).


Noteheads.
http://www.noteheads.com (Acc. 01-09-30).

Valença, Paula (2000). *Developing with plugins and components*