Nuottipiirturi - Loppudokumentti

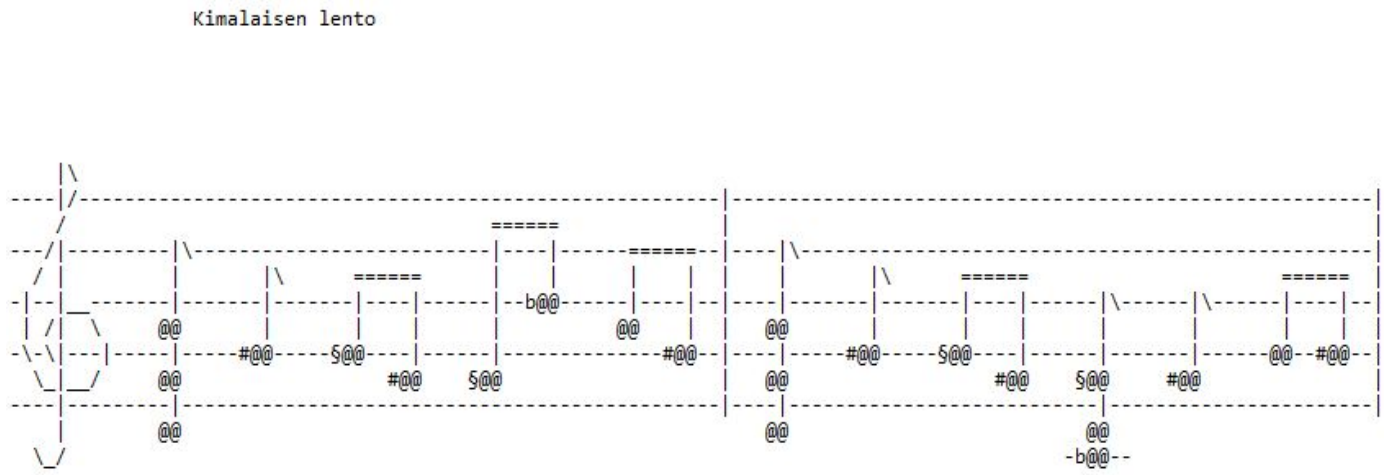
kurssille Ohjelmointistudio2 (MOOC)

Niina Saarelainen
ei opiskelijanumeroa (koulutus: pianonsoitonopettaja ja musiikkiteknologian maisteri)
26.4.2017

1. Ohjelman yleiskuvaus

Ohjelmalla voidaan kirjoittaa yksinkertaisia sävelmiä nuottiviivastolle ja esittää nuotit tekstimuodossa. Ohjelmalle syötetään halutut nuotit tiedostosta tekijän itsensä suunnittelemassa formaatissa. Graafinen ulkoasu on oleellisesti sama kuin mallissa: https://plus.cs.hut.fi/studio_2/2017/studioprojekti/283/. Pieni ero on siinä, että minulla pidemmät nuotit saavat enemmän tilaa ympärilleen, eli kuvastavat nuotin kestoa graafisesti.

Alla ohjelmalla tuotetun kappaleen alku: kappaleen nimi ja ensimmäinen viivasto. Lisää nuottiesimerkkejä liitteessä 2, dokumentin viimeisellä sivulla.



Yllä olevan nuotinnuksen tekoon tarvitaan alla oleva teksti- eli syötetiedosto:

```
#nimi Kimalaisen lento
    <a1,f1,d1> g#1  g1 f#1   f1 b1    a1   g#1          <a1,f1,d1> g#1  g1 f#1  <f1,d1,cb1> f#1  g1 g#1
```

Projektisivulla olevista vaatimuksista toteutettiin kaikki paitsi tahtirajan ylittävät nuotit ja joitakin aika-arvoja, aivan kuten yleissuunnitelmassa lupasin. Lisäksi on toteutettu kappaleen soittaminen GeneralMIDI-instrumentteja käyttäen, mikä mielestäni on hieno piirre: käyttäjä kuulee heti onko hänen ajattelemansa sävelkulku hyvä tai onko syötetiedostossa oleva data se mitä hän kuuli päässään. Nuottien kirjoittaminen oikein ilman kuulokuvaa on haasteellista jopa musiikin ammattilaiselle, harrastelijoista puhumattakaan. Soitto-ominaisuus on kaikissa nuotinnusohjelmissa. Tämän ominaisuuden toteuttamista pidin tärkeämpänä kuin esimerkiksi laajempaa aika-arvojen valikoimaa. Nuottirivit eli viivastot skrollautuvat oikea-aikaisesti kappaleen soidessa.

Ohjelmassa on kiinnitetty huomiota syötteen oikeellisuuden tutkimiseen ja sen korjaamiseen. Tällä estetään ohjelman kaatuminen tilanteissa, joissa käyttäjä ei ole jaksanut kunnolla lukea käyttöohjeita, ei ole tajunnut niitä tai tekee kirjoitusvirheitä syötetiedostoa tehdessä.

2. Käyttöohje

Ohjelman käynnistystiedosto on NuottiPiirturi-projektin main-kansiossa > (default package) > **main.scala**. Tiedostolle sanotaan Run > As Scala Application (Shift F11)

Valmiin kappaleen saa nuoteiksi kirjoittamalla tiedoston nimi tai copy-pastaamalla tiedostolistauksesta ja painamalla ENTER. Tämän jälkeen tulee valita soundinumero 1-7 tai pelkkä ENTER.

Jos haluaa itse tehdä syötetiedoston, saa valinnalla 1 seuraavat ohjeet konsolille. Eli alla projektin juuresta löytyvän help.txt:n sisältö:

O H J E E T :

Ihan ensiksi luo uusi tiedosto input -hakemistoon (hiiren oikealla New > File).
Tiedostopäätteeksi .txt tai ei päätettä ollenkaan.

Sitten voit ruveta kirjoittamaan nuotteja G-avaimelle. Alin mahdollinen ääni on keski-c, ylin kaksiviivainen h. Erottele äänet toisistaan välilyönnillä (saa olla useitakin). Tahtiviivoja ei merkitä.

Rivinvaihtoja saa käyttää vapaasti. Rivinvaihto ei vaihda riviä nuottiviivastolla, eli käytä rivinvaihtoa esim. tahdin tai fraasin lopussa, kunhan itse tiedät miten löydät haluamasi kohdan kappaleessa.

Tässä esimerkki kuinka kirjoitat Ukko Nooan alun alkaen keski-c:stä:
c1- c1- c1- e1-
d1- d1- d1- f1-
e1- e1- d1- d1-
c1----

Sama oktaavia ylempää, rivinvaihtojen puuttuminen ei muuta nuottikuvaa:
c2- c2- c2- e2- d2- d2- d2- f2- e2- e2- d2- d2- c2----

Viivojen (--) määrä viittaa pituuteen eli neljäsosanuotti on yhden iskun eli yhden viivan mittainen c1-

Tässä kaikki käytettävissä olevat pituudet:

```
                |         |                     |          |          |
                |         |                     |          |          |
() = ----    () = --  (). = --- tai --.    @@ = -    @@. = -.  @@
```

Kokonuotti puolinuotti pisteellinen puolinuotti neljäsosa pist.neljäsosa ⅛-nuotti = (ei viivoja)

Etumerkit: ylennä nuotti näin: g#1, alenna näin: gb1. Älä siis kirjoita gis tai ges.
Palautusmerkkejä ei tarvita, eli jos on yksiviivainen fis ja f ovat peräkkäin, kirjoita f#1 f1.

Tauon symboli on z. Et tarvitse muita merkkejä. Perään muista laittaa pituus, esim.
Puolitauko, eli kahden mittainen tauko merkitään z--

Muista aina oktaaviala (1 tai 2), muutoin ohjelma ei tiedä, kumpaa nuottia tarkoitat.

Kannattaa katsoa mallia valmiista kappaleista, eli voit tuplaklikata auki minkä tahansa input-kansion tiedoston.

_____ JO NÄILLÄ OHJEILLA KANNATTAA KOKEILLA OHJELMAN KÄYTTÖÄ, _____
_____ JOS OLET ENSIKERTALAINEN ! _____
_____ SEURAAVAT OPTIOT EIVÄT OLE PAKOLLISIA _____

_____ ALLA LISÄOHJEITA EDISTYNEILLE:_____

Soinnut merkitään <>-merkkien sisään pilkuilla erotettuina. Esimerkkeinä
c-molli kahdeksasosina: <c1,eb1,g1>
G7 puolinuotteina: <g1--,h1--,d2--,f2-->

Soinnussa voi olla ääniä kaksi tai enemmän. Soinnun äänten tulee olla yhtä pitkiä keskenään.
Muista pilkut !

Jos haluat kappaleeseen sanat, toimi seuraavasti:
1) tee ensin kaikki nuottidata (nuotit, tauot, soinnut) eli biisi valmiiksi
2) seuraavalle riville tunniste #sanat ja sen perään sanoja, joko uudelle tai samalle riville, tavutettuina:
Jaak-ko kul-ta Jaak-ko kul-ta

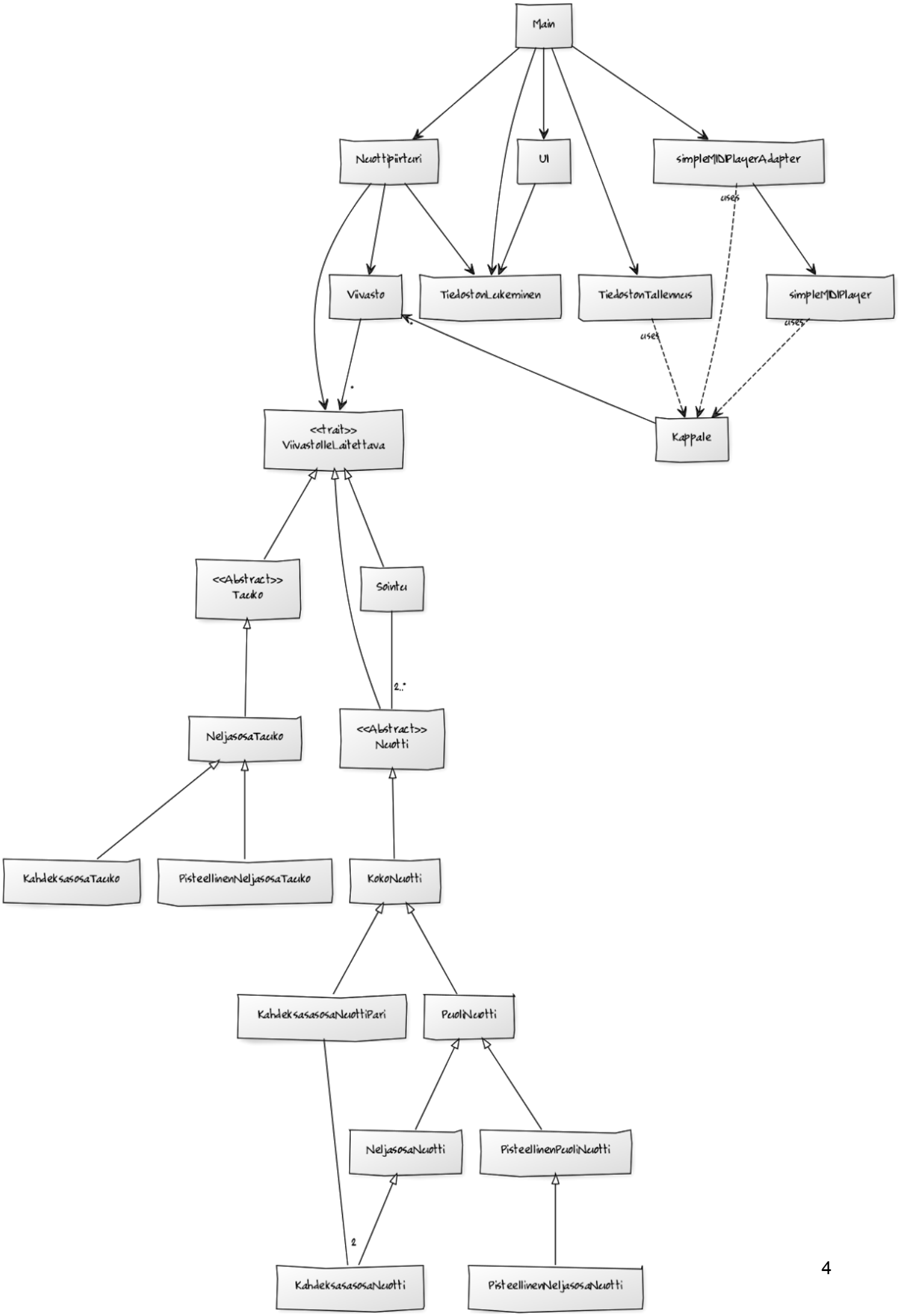
Jos haluat kappaleelle nimen, kirjoita esim.
#nimi Säkkijärven polkka tai
#NIMI Sibelius: Finlandia (isoilla ja pienillä kirjaimilla ei ole väliä tunnisteissa)

Jos kappale ei mene 4/4-tahtilajissa, tulee merkitä (jonkin) rivin alkuun moneenko lasketaan seuraavasti:
#3 (kolmijakoinen) tai vaikkapa #5 (viisijakoinen)

Huomaa siis tunnisteen # käyttö, jos teet nimen, tahtilajin tai sanat.
Kaikki edellä mainitut voi myös jättää pois.

Ylimääräisiä kommentteja voi tehdä tyyliin
kertosäe tai
#sopisikohan tähän paremmin c2?

Esimerkkinä kappaleesta, jossa paljon tunnisteita voit avata input-kansiosta tiedoston FürElise.


Main
Nuottipiirtäjä
UI
simpleMIDIPlayerAdapter
uses
Viivasto
TiedostonLukeminen
TiedostonTallennus
simpleMIDIPlayer
uses
uses
*
*
<<trait>>
ViivastolleLaitettava
Kappale
<<Abstract>>
Taikko
Sointu
2..*
NeljasosaTaikko
<<Abstract>>
Nuotti
KahdeksasosaTaikko
PisteellinenNeljasosaTaikko
KokoNuotti
KahdeksasasosaNuottiPari
PuoliNuotti
NeljasosaNuotti
PisteellinenPuoliNuotti
2
KahdeksasasosaNuotti
PisteellinenNeljasosaNuotti

3. Ohjelman rakenne

Edellisellä sivulla ohjelman UML-kaavio.

Ohjelmaan on tullut teknisessä suunnitelmassani esitetyn luokkajaon jälkeen seuraavat muutokset:

- Lisätty luokat: ***Main, UI, SimpleMIDIPlayerAdapter, SimpleMIDIPlayer***
- Poistettu turhana luokka: ***SoivaSymboli***

Aiemmin ajattelin, että ***NuottiPiirturi*** ja ***TiedostonLukeminen*** "pääluokkina" voisivat hoitaa ohjelman flow:n. Kuitenkin kiitos FT Matti Luukkaisen, joka huomautti että koko ohjelman olisi hyvä toimia luuppina (eli että voi printata useita kappaleita samalla käynnistyskerralla), totesin että ***Main*** on syytä olla jakamassa käskyjä muille luokille ja ohjelmasilmukka on hyvä olla muualla kuin esim. luokassa ***NuottiPiirturi***, jolla on jo muitakin tehtäviä.

UI (User Interface) perustettiin hoitamaan käyttäjältä saatava syöte. Aluksi tämä toiminta oli useassa luokassa, siinä kohdassa koodia kun oli käyttäjäinteraktion vuoro. Kuitenkin älykkäämpää on keskittää toiminta yhteen luokkaan ja varsinkin yksikkötestejä tehdessä tajusin, että testejä ei voi automatisoida, jos ohjelma odottaa käyttäjän syötettä kesken testin.

Nämä suunnitelmaan tehdyt muutokset ovat Single responsibility -periaatteen mukaisia.

Main on ohjelman logiikkakeskus. Se on yhteydessä useisiin muihin luokkiin.
Tärkeimmät metodit:

- ohjelmanRunko() ohjelman ydintoimintojen kutsuminen
- rekursiivinen kelvollisenTiedostonKysyminenJaTarkistusLoop() pyytää ***UI***-luokkaa tutkimaan onko käyttäjän valitsema tiedosto hakemistossa ja kysyy uutta jos ei ole. Jos on, kutsuu ***TiedostonLukeminen***-luokan metodia lueTiedosto. Jos tiedostossa ei ole nuottidataa, aloitetaan tämän metodin toiminta alusta
- valitseToiminto() antaa käyttäjän valita seuraava toimenpide tai lopettaa

UI huolehtii käyttäjältä tulevista syötteistä: *Syötetiedoston nimi*, *MIDIPatch*, eli millä soundilla kappale soitetaan ja *talletettavan tiedoston nimi*.
Tärkeimmät metodit:

- listaaTiedostot() kertoo mitä tiedostoja kohdehakemistossa on tällä hetkellä
- kayttajaValitseeTiedoston(lukija: TiedostonLukeminen) kysyy toiselta metodilta onko kyseistä tiedostoa olemassa
- kayttajaValitseeMIDIPatchin(): käyttäjälle annetaan mahdollisuus valita 7 eri soundia tai myös voidaan katsella nuottikuvaa hiljaisuudessa.
- kayttajaValitseeTiedostonTallennusnimen(): käyttäjä saa valita nuotinnoksen tallennusnimen tai olla tallentamatta.
- mitaTehdaanSeuraavaksi() antaa mahdollisuuden lukea käyttöohjeet, valita uusi tiedosto tai lopettaa

TiedostonLukeminen -luokassa tapahtuu käyttäjän valitseman tiedoston lukeminen ja tiedon jatkokäsittely. Jos syötettä oli, se pitää tarkistaa ennen kuin se kannattaa lisätä nuottiAlkioihin, jotka sisältävät pelkästään oikean syntaksin omaavia nuotteja, taukoja ja sointuja String-olioina.

Tärkeimmät metodit:

- lueTiedosto(tiedostonNimi) lukee käyttäjän ***UI***:ssa valitseman tiedoston rivi kerrallaan puskuriin. Jos tiedosto on tyhjä tai jos tiedoston sisältö on esim. `#nimi John Cage: 4’33”` (eikä nuottidataa), palataan returnilla takaisin kelvollisenTiedostonKysyminenJaTarkistusLoop -metodiin luokassa ***Main***. Jos dataa on, kutsutaan ensin kasitteleTunnisteet() -metodia, sitten tarkistaVirheet() -metodia.
- kasitteleTunnisteet() lajittelee #-merkintöjen perusteella syötetiedoston dataa. Syötteet jaetaan viiteen kategoriaan: 1) kappaleen nimi, 2) tahtilaji, 3) lyriikat, 4) vapaaehtoiset kommentit ja 5) nuottidata, eli viivastolle päätyvät nuotit, tauot ja soinnut.
- tarkistaVirheet() splittaa syötetiedoston välilyönnistä alkioiksi, jotka pitää kaikki tutkia virheiden varalta. Epäkelvosta syötteestä käyttäjä saa virheilmoituksen, jossa kuvaillaan virheen tyyppiä ja kerrotaan millä rivillä se on. Kun käyttäjä on korjannut sen, kutsutaan lueTiedosto:a uudestaan. Sieltä ohjelma palaa tarkistamaan virheitä niin kauan kuin niitä on.
- oikeellisuusTesti(syote: String) tutkii kaikkien alkioiden (“c#2--.”) nimeen (“c#2”) ja pituuteen (“--.”) liittyvän tiedon ennenkuin se lisätään nuottiAlkiot-puskuriin. Virheellinen syöte (esim. “c”) tuottaa käyttäjälle printtautuvan virheilmoituksen: “tarkoititko c1 vai c2? Virhe on rivillä 12. Korjaa äsken valitsemaasi tiedostoon ja paina ENTER, kun tiedosto on tallennettu.”

NuottiPiirturi:ssa tapahtuu nuottidatan ja lyriikoiden käsittely ja ohjeiden antaminen muille luokille. Tärkeimmät metodit:

- execute() päämetodi, joka kutsuu luokan muita metodeja ja luomaansa ***Viivasto***-oliota
- rekursiivinen kasitteleNuottiTieto(), jonka toiminta selitetään luvussa Algoritmit
- tutkiEtumerkit() päättää tarvitaanko palautusmerkkiä tai jätetäänkö etumerkki piirtämättä
- kasitteleLyriikat() splittaa lyriikat välilyönnistä ja kaikki paitsi tyhjät merkkijonot lisätään puskuriin
- tehdaanKahdeksasosaParit() muodostaa yksittäis-⅛-olioista tietyillä ehdoilla kahdeksasosa-pareja jättäen muut nuottioliot ennalleen.

Viivasto piirtää ***ViivastolleLaitettava-***olioiden kuvakenttiä ja jos kappaleessa on sanat, ne kirjoitetaan synkroonissa kuvien kanssa. Kun yksi rivi on valmis, lisää ***Viivasto***-olio itsensä ***Kappale***-olion loppuun. ***Kappale-***olio sisältää kappaleen nimen ja sen jälkeen nuottiviivastoja, eli ***Viivasto***-olioita

Tärkeimmät ***Viivasto***-luokan metodit:

- def piirraNuotit() kutsuu muita metodeja, kun nuottiolion kuva liitetään edellisiin, kun on aika piirtää tahtiviiva tai vaihdetaan riviä
- kasitteleLyriikat() kahdeksasosaparin kohdalla luetaan kaksi tavua eli lyriikkapuskurin 2 seuraavaa indeksiä, muutoin yksi. Tavu(t) tallennetaan nuottiolion kuva-kentän alimmalle riville.

SimpleMIDIPlayerAdapter käyttää ***NuottiPiirturi:***ssa luotua nuottiData-puskuria (tyyppiä Buffer[ViivastolleLaitettava]). Puskurin nuottiolioiden korkeus-kentät muunnetaan MIDI-spesifikaation mukaisiksi MIDI-numeroiksi ja nuottiolioiden pituus-kenttä käy sellaisenaan määrittämään oikean mittaisen nuotin millisekunneissa. MIDI-numero, tai soinnun tapauksessa MIDI-numerot ja pituus zipataan yhteen ja lähetetään ***SimpleMIDIPlayer***:lle yhdessä MIDIPatch:n, kappale-olion ja tahtilajitiedon kanssa.

SimpleMIDIPlayer soittaa kappaleen käyttäjän valitsemalla saundille ja luokan vastuulla on myös kappaleen nuottidatan esittäminen rivi kerrallaan musiikin kanssa oikea-aikaisesti. Luokka printtaa ruudulle ***Kappale***-oliota ***Viivasto***-olio kerrallaan. Skrollaushetki selviää laskemalla ms-muuttujaa ja tahtilajitietoa hyödyntäen rivin viimeisen tahdin viimeisen iskun ajankohta. Skrollaus tapahtuu siis

ennakoiden, jotta uusi rivi on silmien edessä ennenkuin se soitetaan, muutoin ensimmäistä nuottia ei ehdi tajuta kun se jo soi. Jos kappaletta ei soiteta, printataan koko kappale kerralla ruudulle ***Main***:ssa.

TiedostonTallennus-olio tallentaa ***Kappale***-olion sisältämät viivastot rivi kerrallaan tiedostoon käyttäjän haluamalla nimellä.

ViivastolleLaitettava -piirreluokka ja sen perivät luokat

Kaikki nuotit, soinnut ja tauot periytyvät trait ***ViivastolleLaitettava***:sta. Yhteistä kaikille on *kuva*, *pituus* eli nuotin/tauon/soinnun kesto[1] ja *soiva*-kenttä, jota käytetään laitettaessa lyriikoita paikoilleen ***Viivasto***- oliossa. Esimerkiksi *korkeus* ei ole yhteinen piirre, koska soinnulle ei voi määritellä vain yhtä korkeutta. Tärkein metodi: piirraTyhjaViivasto(leveys:Int) piirtää viisi parametrina saadun mittaista viivaa, eli tietyn levyisen palan nuottiviivastoa, johon kukin nuotti-/tauko-olio myöhemmin piirtää itsensä.

Abstrakti ***Nuotti-***luokka perii ***ViivastolleLaitettava***:n ja lisää kaikille nuoteille yhteisiä ominaisuuksia: *nuppi*, nuotin *korkeus* ja *etumerkkiin* liittyviä määritteitä. piirraAlaApuviiva() ja piirraYlaApuviiva() -metodeita kutsutaan, jos piirrettävä nuotti on keski-c, a2 tai h2, tai näiden ylennetty tai alennettu muoto.

Luokka ***KokoNuotti*** on kaikkien nuottien isä. Täällä abstraktissa Nuotti-luokassa esitellyt metodit saavat toteutuksensa ja näitä metodeja käyttävät kaikki Nuotti-oliot. ***PuoliNuotti*** ja kaikki sen perivät nuotit ovat varrellisia, joten metodi piirraVarsi() tulee mukaan tässä vaiheessa perimisjärjestystä. ***PisteellinenPuoliNuotti*** on kaikkien pisteellisten nuottien isä. Luokka kutsuu PuoliNuotin kuva-metodia ja lisää nuotin perään pisteen. ***NeljasosaNuotti***:ssa vaihdetaan nupin grafiikka, määritellään pituudeksi 1.0 ja muutetaan kuvanLeveyttä vastaamaan nuotin viemää kestoa graafisesti. Muutoin käytetään yläluokkien määrittelyjä ja metodeja. ***KahdeksasosaNuotti***:ssa piirretään NeljasOsalta perittyyn kuvaan väkä.

KahdeksasosaNuottiPari perii yllättäen ensimmäisen nuotin Kokonuotista, koska kaikki muiden luokkien antamat “varustukset” tulisi kuitenkin purkaa: esim. yksittäistapauksessa neljäsosanuotin (potentiaalinen isä) varsi voi olla alaspäin, mutta kahdeksasosaparissa piirretään uudella logiikalla ja varsi voi mennä eri suuntaan. Toinen nuotti piirretään tämän luokan ohjeilla, koska sen sijainti on täysin uusi x-akselilla (ei esi-isiä). Varret ja palkki piirretään vertaamalla kummankin nuotin korkeutta suhteessa piirtoalueen reunoihin ja piirretään varret sinne missä on enemmän tilaa.

Luokka ***Sointu*** on kokoelma ***Nuotti***-olioita***.*** Jokaisen nuottiolion nuppi piirtyy paikalleen käyttäen esi-isiltään perittyä tietotaitoa. Varren pituus pitää laskea kullekin soinnulle erikseen, pituus riippuu alimman ja ylimmän äänen sijainneista. Yläapuviiva ja ⅛-nuottien väkä vaativat myös omat laskutoimenpiteet (ei voi periä muualta) varren suunnasta ja pituudesta johtuen.

Abstrakti ***Tauko***-luokka lisää ***ViivastolleLaitettava***:n ominaisuuksiin vain vakiopiirtokorkeuden.

[1] Nyt vasta dokumenttia kirjoittaessa tajusin että kesto olisi ollut huomattavasti parempi nimi kentälle kuin pituus. Pituus sanana voi viitata grafiikkaan, mutta kesto olisi ollut täydellinen, musiikille uniikki termi.

4. Algoritmit

Esitellään 4 algoritmia, jotka liittyvät nimenomaan nuottitiedon käsittelyyn. Vastakohtana tässä ajattelen algoritmeja, jotka voisivat olla muissakin ohjelmissa, esimerkiksi syötteen lajittelu tunnisteiden perusteella tai tiedoston lukeminen ja tallentaminen. Jätän käsittelemättä myös lyriikat, koska tekstin käsittely on triviaalimpaa kuin esimerkiksi sointujen.

#1 Buffer[ViivastolleLaitettava]:n muodostaminen Buffer[String]:stä

Tässä selitetään pseudokoodina rekursiivisen metodin kasitteleNuottiTieto toiminta luokassa NuottiPiirturi. Jos inputBuffer:n sisältö on: “<c1,e1,g1>”, “c2” (eli yksi sointu ja yksi nuotti), kyseistä metodia kutsutaan 5 kertaa: kerran soinnulle, 3 kertaa soinnun sävelille ja lopuksi vielä kerran sävelelle c2. Lopputuloksena metodi tuottaa Buffer(Sointu, Nuotti)-tyyppisen puskurin.

```
def kasitteleNuottiTieto (inputBuffer: Buffer[String],  palautetaan: Buffer[ViivastolleLaitettava] ):
Buffer[ViivastolleLaitettava] = {
    Joka alkiolle {
        jos alkion ensimmäinen merkki on ‘<’, niin kasitteleSoinnut(alkio)
        muuten {
                muodosta nuotinNimi poistamalla alkiosta pituustieto
                laske ‘-’-merkkien määrä pituus-muuttujan arvoksi
                muodosta mahdollinen extraetumerkki
                jos kyseessä on tauko {
                        muodosta oikea määrä Tauko-olioita tutkimalla pituus-muuttujaa...
                        ...ja alkion mahdollista  ‘.’-merkkiä (pisteellisten nuottien tunnus)
                        talleta palautetaan-puskuriin
                        kasvata iskujaMennyt[2] -muuttujaa tauon pituudella
                } muuten kyseessä on nuotti, jolloin {
                        muodosta oikea Nuotti-olio tutkimalla pituus-muuttujaa...
                        ... ja alkion mahdollista ‘.’-merkkiä
                        talleta palautetaan-puskuriin
                        kasvata iskujaMennyt-muuttujaa nuotin pituudella, jos kasvataIskuja[3] >= 0
                }
        jos iskujaMennyt-muuttujan arvo on kasvanut samaksi kuin tahtilaji-muuttujan arvo {
                iskujaMennyt nollataan
                tahdinAikaisetEtumerkit-puskuri nollataan
        }
        kasvatetaan iskujaMennyt -muuttujan arvoa yhdellä kaikille alkioille
    }


        // sisäkkäinen metodi, jota kasitteleNuottiTieto kutsuu:
```

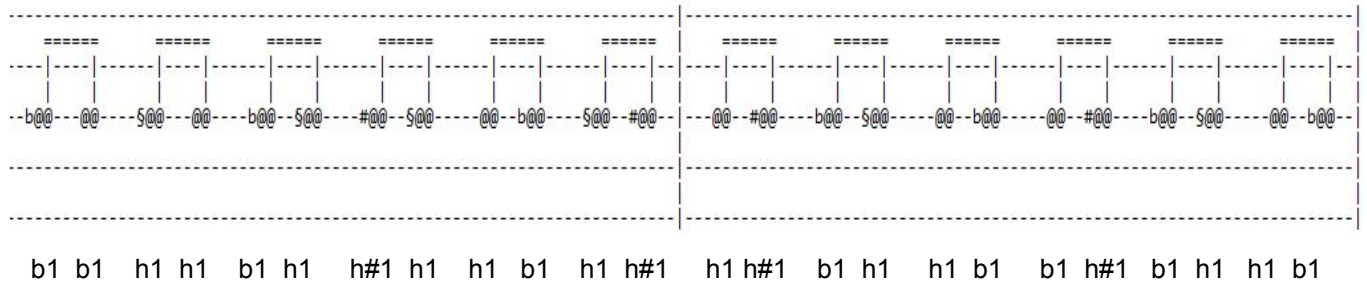
[2] Muuttujat *iskujaMennyt...*

[3] ...ja *kasvataIskuja* on määritelty metodin ulkopuolella, koska rekursiosta johtuen niiden arvot nollautuisivat aina uudella rekursiokierroksella, jos ne alustettaisiin itse metodissa. Nämä muuttujat pitävät yllä tietoa siitä, millä tahdinosalla ollaan menossa. Soinnun tapauksessahan ei saa kasvattaa iskujen määrää kuin kerran, ei jokaisen soinnun sävelen kohdalla. (Soinnuissa kaikki äänet ovat yhtä pitkiä keskenään.)

```
def kasitteleSoinnut(alkio: String) = {
        laita uuteen Buffer[String] -tyyppiseen muuttujaan soinnun sävelten nimet
        luo tyhjä Buffer[ViivastolleLaitettava] -tyyppinen muuttuja
        kasvataIskuja saa arvon, joka on 0 - (sointujen äänten määrä + 1)
        Nuottipiirturin muuttujaan nuottiData lisätään Sointu-olio, joka muodostuu kun...
        ... kutsutaan kasitteleNuottiTieto:a äsken luoduilla muuttujilla
}
```

Aivan *kasitteleNuottiTieto*-metodin lopuksi palautetaan *palautetaan*-puskuri, jonka sisältö on joko...
... Sointu-olio tai kokoelma Nuotti-, Tauko- tai Sointuolioita, molemmat tyyppiä Buffer[ViivastolleLaitettava]

#2 Etumerkkilogiikka eli metodin tutkiEtumerkit() toiminta



b1 b1 h1 h1 b1 h1 h#1 h1 h1 b1 h1 h#1 h1 h#1 b1 h1 h1 b1 b1 h#1 b1 h1 h1 b1

Yllä on malliesimerkki, mitä etumerkkilogiikalla tarkoitetaan: Etumerkki on voimassa koko tahdin, ellei toinen samaan nuottiin kohdennettu etumerkki kumoa sitä. Kuvassa toisen nuotin kohdalla huomataan, että nuotin nimi on b1, mutta etumerkkiä ei piirretä. Mikään sama etumerkki ei siis toistu nuottikuvassa peräkkäin, koska se jää yhdestä kerrasta voimaan. Tahtiviiva kumoaa kaikki etumerkit, siksi yllä toisen tahdin ensimmäisessä nuotissa ei saa olla palautusmerkkiä, vaikka edellisessä nuotissa on ylennys.

Ohjelmassani etumerkkejä on kahta kategoriaa: varsinaisina etumerkkeinä pidän ylennys- ja alennusmerkkiä, jotka ovat osa nuotin nimeä. Tämä ei vielä riitä piirtämiseen: nuotin nimessä oleva etumerkki ei näy esim. nuottiesimerkin toisessa nuotissa: tähän tarvitaan toinen parametri -extraetumerkki - kertomaan, että merkkiä ei saa piirtää, olen itse keksinyt lyhyen merkinnän “n” (“neutraalin näköinen nuotti”). Joku voisi tässä oikoa asioita, ja kutsua alennusmerkitöntä b:tä h:ksi, mutta päätin että en tee niin, koska se on musiikinteoreettisesti väärin ja lisäksi kappaleen soittaminen menee väärin, jos vaihdan nuotin nimen. Eli en nimeä nuottiesimerkin toista nuottia h:ksi vaikka se h:lta näyttää (vrt. miten 4. nuotti h1 piirretään). Samoin kaikki palautusmerkit ilmoitetaan extraetumerkissä, koska se ei ole samassa asemassa kuin b ja # : nuotin nimessä ei ikinä näy palautus, koska syötteessä ei tule vastaan nuotteja kuten c§1 tai f§2.

Toteutan tämän ylläpitämällä puskuria, johon laitan tahdin ajaksi kaikki ylennettyjen tai alennettujen nuottien nimet. Sitä pitää tutkia jokaisen nuotin piirtohetken kohdalla, myös ylentämättömien, koska kuten yllä olevasta esimerkistä huomataan, joskus h1 saa palautusmerkin, joskus ei.

Puskuriin ei pelkästään laiteta nykyistä ylennys- tai alennusmerkillä varustettua ääntä, vaan sieltä pitää myös poistaa saman nuotinkorkeuden eri versiot. Eli jos nuotti oli aluksi alennettu, ja nyt se ylennetään, pitää alennettu nuotti poistaa puskurista. Jos nuotti palautetaan, otetaan sen mahdollinen alennettu tai ylennetty versio pois puskurista. Jos tämän poistovaiheen tekee huolimattomasti, käy niin,

että palautusmerkkejä piirtyy turhaan, esimerkiksi sekvenssissä: c#1, cb1, c1, c1 tulisi molemmille c1:lle palautusmerkki, jos puskurista ei myös poisteta vanhaa tietoa.

Suurin osa nuottien nimistä rakentuu samalla kaavalla, esim. c#1 ja gb2 ovat samaa tyyppiä: 1) kirjain, 2) etumerkki, 3) oktaaviala. Ilman etumerkkiä nimi on tyyppiä 1) kirjain, 2) oktaaviala. Eli näissä tapauksissa ensimmäinen ja viimeinen merkki kertovat nuotin “perussävelen” (hieno musiikkitermi on juurisävel). Ikävän poikkeuksen tekee nuotin nimi b, jossa juurisävel h ei näy, sen sijaan tieto alennuksesta ja nuotin nimestä ovat molemmat ensimmäisessä merkissä. Siksi algoritmissa on 6 erillistapausta tätä nimeämisen epäloogisuudesta johtuvaa asiaa huomioimassa.

Tahtiviivalla ylennettyjen ja alennttujen nuottinimien sisältämä puskuri nollataan. Tahtiviivan paikan laskeminen kuuluu metodille kasitteleNuottiTieto, josta metodia tutkiEtumerkit() kutsuttiin.

#3 Luokan KokoNuotti kuva-kentän piirtäminen

KokoNuotti on kaikkien muiden Nuottien kantaluokka. Itse kuva-kentän koodi on kaksi riviä:
piirraNuppi(); viivasto

viivasto on piirreluokalta ViivastolleLaitettava peritty Buffer[String]-tyyppinen muuttuja, jonka leveys kokonuottien tapauksessa on 22. Eli KokoNuotti-oliota luodessa piirretään tyhjä viivasto, jossa jokainen viivaston viiva ja väli on 22 merkin mittainen.

Selitetään pseudokoodilla piirraNuppi():

> jos piirrettävä nuotti on keski-c tai sen etumerkillinen muoto, piirraAlaApuviiva()
> jos piirrettävä nuotti on joku näistä: "a2", “ab2”, “a#2” "h2", “h#2”,"b2", "bb2", piirraYlaApuviiva()
> korvaa viivasto-muuttujan tietyssä indeksissä oleva merkkijono uudella merkkijonolla, johon piirrä...
> ...mahdollisesti etumerkki ja vähintään nuotin nuppi.

Lopuksi palautetaan viivasto, jossa on oikean mittainen pala viivastoa ja siinä kokonuotti

#4 Syötteen oikeellisuuden tarkistaminen

Syöteformaatti on tavallaan yksinkertainen, mutta käyttäjätestauksessa havaitsin, että joitakin asioita ei jaksettu katsoa tarkasti. Kun ohjeessa luki: merkitse sointu näin: <c1,d1,f1> kirjoitti käyttäjä kuitenkin soinnun tähän tyyliin: <c1, d1, f1> Erona siis välilyönnit. Koodi muokkaa datan speksien mukaiseksi niissä tapauksissa kun mielestäni voidaan varmuudella sanoa, mitä käyttäjä tarkoitti.

Joissakin tapauksissa kuitenkin käyttäjää pyydetään korjaamaan virheet, eli tapauksissa joissa ei ole tarpeeksi tietoa siitä mitä tarkoitettiin. Jos käyttäjä kirjoittaa nuotin nimeksi c, ohjelma ei rupea arvaamaan tarkoitettiinko c1 vai c2, vaan sen joutuu käyttäjä korjaamaan syötetiedostoon.

Tässä lista, mitä kaikkia vastoin speksejä kirjoitettuja asioita koodi korjaa itsenäisesti:

- replaceAll-komennoilla korvataan: Soinnun sisältä välilyönnit pois, kaikki enemmän kuin yksi välilyöntiä korvataan yhdellä, tyhjät soinnut pois (hyvin epätodennäköistä, että käyttäjän syötteessä olisi “<>”, mutta varmuudeksi poistetaan tämäkin ohjelman kaatamismahdollisuus), kaksi sointua kirjoitettu peräkkäin ilman välilyöntiä → välilyönti, eli "><" → "> <", " -" korvataan "-":lla, eli pituustieto pitää olla kiinni nuotissa, tabulaattorimerkit pois.

- Jos alkiossa on kaikki elementit, mutta väärässä järjestyksessä, laitetaan ne oikeaan järjestykseen, eli oikeellisuusTesti()-metodi hyväksyy kaikki seuraavat: c#1- , c1#- , c1-# , -c1#, c-1# tarkoittamaan yhden mittaista ääntä c#1- eli yksiviivainen neljäsosa-cis.

Tässä pseudokoodina kaikki ne asiat, joista tulee virheilmoitus, eli asiat, jotka käyttäjän tulee korjata. Muuttujien nimet on kursivoitu.

def oikeellisuusTesti (syote: String): String {
poistetaan syötteestä kaikki pituuteen viittaava tieto (merkit '-' ja '.') ja talletetaan *filtteredNote*:een
jos *filtteredNote* on "z" (eli tauko), ei palauteta virheilmoitusta
muuten {
jos *filtteredNote* on tyhjä merkkijono , palauta **"pelkkä pituustieto, puuttuu nuotin nimi"**
jos *filtteredNote* sisältää useamman 'z':n, palauta **"taukojen pituudet merkitään viivoilla, esim... ...puolitauko z--"**
jos *filtteredNote:*n ensimmäinen merkki pieneksi kirjaimeksi muunnettuna ei löydy merkkijonosta... ..."cdefgahb", niin palauta **"nuotin pitää alkaa kirjaimilla c,C, d,D e,E f,F g,G a,A h,H, b tai B"**
jos *filtteredNote:*n pituus on 1, palauta **"tarkoititko "+ *filtteredNote*+ "1 vai " + *filtteredNote*+ "2?"**
jos *filtteredNote:*n pituus on 2 ja jos *filtteredNote.tail* sisältää ylennys- tai alennusmerkin, mutta ei ... ...sisällä "1":stä tai "2":sta, palauta **"tarkoititko "+ *filtteredNote*+ "1 vai " + *filtteredNote*+ "2?"**
jos *filtteredNote* sisältää "#b":n tai "b#":n, palauta **"nuotissa on ylennys- ja alennusmerkki"**
jos *filtteredNote* ei sisällä "1":stä tai "2":sta, palauta **"nuotissa tulee olla oktaaviala: 1 ja 2"**
jos *filtteredNote:*n pituus on 3 ja jos *filtteredNote.tail* ei sisällä ylennys- tai alennusmerkkiä, palauta **"väärä formaatti. Muistathan syntaksin: esim. alennettu e on Eb, ei es"**
jos *filtteredNote:*n pituus on yli 3, palauta **"liian pitkä nuotin nimi"**
}

Sitten siirrytään tutkimaan pituuteen liittyviä mahdollisia virheitä syötteessä:

lkm-muuttuja saa arvokseen syötteessä olevien viivojen lukumäärän
jos *lkm* on suurempi kuin 4, palauta **"maksimipituus nuotille on 4, eli viivoja korkeintaan ----"**
jos *lkm* on 3 ja syöte sisältää pisteen, palauta **"väärä pituus"**
jos *lkm* on 4 ja syöte sisältää pisteen, palauta **"pisteellistä kokonuottia ei ole määritelty, tee... ...kokonuotti ja tauko"**
jos lkm on 0 ja syöte sisältää pisteen, palauta **"tämä ohjelma ei osaa käsitellä pisteellistä... ...kahdeksasosaa"**

Lopulta voidaan palauttaa tyhjä merkkijono, jos syöte ei jäänyt kiinni mistään edeltävistä rikkeistä
}

5. Tietorakenteet

Esitellään 4 tärkeää tietorakennetta:

1) **Buffer[ViivastolleLaitettava]** luokassa NuottiPiirturi :
Kaikki ohjelmassa piirrettävät nuotti- ja tauko-oliot ovat tyyppiä ViivastolleLaitettava. Oli välttämätöntä kehitellä tällainen kaikille piirrettäville olioille yhteinen luokka, jotta ne voidaan koota saman "katon" alle. Tämä tietorakenne on pohjana grafiikalle ja soittamiselle. Puskurin

kannattaa olla *mutable*, koska sitä luodaan alkio kerrallaan ja ei olisi järkeä luoda uutta puskuria joka lisäyksellä.
Puskuria luetaan useassa kohdassa ohjelmaa ja olioiden kuva- ja pituustietoja käytetään viivastojen muodostamiseen. Soittamiseen tarvitaan korkeus- ja pituustietoa, jotta saadaan MIDI-systeemin tarvitsema MIDI-numero ja nuotin kesto millisekunneissa.

2) **Map-hakurakenteen** käyttö useassa kohtaa ohjelmaa:
Luokista ViivastolleLaitettava, Viivasto ja SimpleMIDIPlayerAdapter löytyy tietorakenne **Map[“nuotin nimi” → Int]**. Nuottien korkeus y-akselilla löytyy, kun avaimeksi annetaan nuotin nimi. Esimerkiksi avain “d1” tuottaa arvon 15, joka tarkoittaa että yksiviivaisen D:n sijainti nuottiolioiden kuva-kentässä on 16. rivillä eli Buffer[String]:n kuudennessatoista merkkijonossa. Viivasto-luokka käyttää samat avain-arvoparit sisältävää Map:iä piirtääkseen tahtiviivat oikeisiin paikkoihin y-akselilla.
SimpleMIDIPlayerAdapter käyttää samaa rakennetta, mutta eri arvoilla. Siellä avain “d1” saa arvon 62, joka on MIDI-spesifikaation mukainen korkeus keski-c:n vieressä olevalle D:lle. Täällä useat avaimet voivat tuottaa saman arvon, kuten musiikissakin käy: sekä d#2 että eb2 saavat arvon 75, koska ne kuulostavat samalta eli ovat saman soivan korkeuden kaksi eri nimeä.

3) Valmis kappale on **Buffer[Buffer[String]]**. Kappaleeseen lisätään uusi viivasto sen jälkeen kun toinen tahtiviiva on saatu piirrettyä, eli on päästy määriteltyyn viivastorivin maksimipituuteen. Viivasto on siis Buffer[String] ja kappale muodostuu yksi viivasto kerrallaan lisäämällä puskurin loppuun. Tämä muuttuva (mutable) tietorakenne on joustavuudessaan hyvä: kaikki sävellykset ovat eripituisia. Lisäksi kirjoittaminen tiedostoon tietorakenteesta **Buffer[Buffer[String]]** on hyvin helppoa kahdella sisäkkäisellä for-loopilla.

4) SimpleMidiPlayerAdapter-luokka muodostaa **Buffer[(Buffer[Int], Double)]-**puskurin, joka sisältää Tuple-pareja tyyppiä (Int-puskuri ja Double). Tämä on kätevä tietorakenne pitämään sisällään kaikki soittamiseen tarvittavat parametrit: **Tuple._1** sisältää soitettavan nuotin korkeuden tai soinnun tapauksessa korkeudet MIDI-numeroina. Tauoille olen itse keksinyt “korkeuden” 0, joka korkeus jätetään soittamatta ja vain pidetään oikean millisekuntimäärän mittainen Thread.sleep-tuokio. MIDI-numero tai numerot kääritään puskuriin, jotta saadaan yhtenäinen tietorakenne. Jos aina olisi vain yksi numero, ei käärimiseen olisi tarvetta, mutta sointujen takia tarvitaan tietorakenne usealle äänelle, joten myös yhden äänen tapauksessa numero kääritään puskuriin.
Tuple._2 on nuotin/tauon keston kerroin (0.5 … 4.0), joka kerrotaan vakiomillisekuntimäärällä[4] ja lopputuloksena on nuotin tai tauon oikea kesto.
Tämä tietorakenne mahdollistaa sen, ettei tarvitse lukea kahta erillistä puskuria ja pidän tyylikkäänä, että koko kappaleen kaikki soittodata on yhdessä paketissa, joka annettaan parametrina SimpleMIDIPlayer-luokalle ja joka muodostettiin siis .zip-metodilla.

[4] Mietin monta kertaa pitäisikö kappaleen temponkin olla käyttäjän säädettävissä oleva parametri. Päädyin siihen, että syötetiedostossa on jo tarpeeksi tunnisteita. Toinen paikka kysyä tempoa olisi joka piirtokerralla konsolilla. Tämä kuitenkin veisi ohjelman painopistettä poispäin tehtävänannosta: grafiikasta ääneen. Ei hyvä.
Käyttäjä voisi myös jossain vaiheessa kyllästyä, jos pitää vastata moniin kysymyksiin ennnenkuin saa nuotit konsolille. (Dokumentin lukijalle tiedoksi: jos haluaa huvikseen testailla kappaleita eri tempoilla, ms-muuttujaa voi säätää SimpleMIDIPlayer-luokassa rivillä 9)

6. Tiedostot

Ohjelma lukee syötteen tekstitiedostosta. Tiedostopääte voi olla .txt, tai tiedosto voi olla päätteetön. .rtf-päätteisiä tiedostoja ohjelma ei osaa lukea oikein: rtf generoi piilodataa, jota ohjelma luulee nuotiksi: {\rtf1\ansi\deff0\nouicompat{\fonttbl{\f0\fnil\fcharset0.

Myös käyttöohjeet sisältävä help.txt on tekstitiedosto. Valmiit nuotit tallennetaan .txt päätteisiksi, jotta tuotoksia voi tarkastella Eclipsellä (esim. rtf:ää ei voi). Ohjelmakoodi lisää .txt-päätteen käyttäjän antamaan nimeen.

7. Testaus

Ohjelmaa on testattu kohtuullisen kattavasti. Oma tavoitteeni oli keksiä joka luokkaan (ohjelmassa on 23 luokkaa) vähintään yksi testi. Käytännössä jokaista ViivastolleLaitettava:n perivää luokkaa ei ole testattu omalla testillä, vaan yhteisellä monen luokan integraatiotestillä: kaikkia nuottityyppejä eli jokaisen mittaista nuottia ja taukoa käytetään kappaleessa, joka tallennetaan tiedostoon ja tiedoston sisältöä verrataan ohjelman toimintaan. Jos koodia tulevaisuudessa muutetaan siten, että muutos aiheuttaa bugin, tämä testi hajoaa helposti. Tekstitiedosto toimii siis referenssinä oikeasta piirtotavasta.

Tärkeät ja paljon koodirivejä sisältävät luokat TiedostonLukeminen ja NuottiPiirturi ovat saanet useita testejä kohdistuen keskeisiin metodeihin ja tietorakenteisiin.

Testeissä on käytetty syötteenä sekä hyväksyttäviä että hylättäviä syötteitä, eli on testattu ohjelman käyttöä syötteillä, joiden pitäisi johtaa onnistuneeseen nuotinnokseen, mutta myös tapauksia, joissa data on virheellistä tai puutteellista. Myös grafiikan reuna-alueita ja reunaehtoja on testattu: mahtuuko varsi piirtymään sekä nuottien että sointujen tapauksissa ylimmässä ja alimmassa nuotissa, mitä tapahtuu liian pitkillä tai liian lyhyillä lyriikoilla, mitä jos tahtiviivalle ei löydy paikkaa, jne.

Testaamista varten jouduttiin koodiin tekemään muutamia muutoksia:

- lueTiedosto()-metodin piti saada parametrinaan tiedoston nimi
- Viivasto-luokkaan tehtiin muuttuja unitTestLiitosCounter vain, jotta voidaan todistaa että tiettyä metodia on kutsuttu oikea määrä. Muutoin metodien toimintaa oli mahdoton tutkia, muuten kuin tallentuneen lopputuloksen kautta, joka tallentuukin jo eri luokan ilmentymään (testaustavoitteeni oli pystyä todentamaan jokin aspekti joka luokassa.)
- SimpleMIDIPlayerAdapter:n kutsuparametrilistaan lisättiin ylimääräinen boolean kertomaan, että kesken yksikkötestauksen ei tarvitse soittaa kappaleita: false-arvo ei kutsu SimpleMIDIPlayer-oliota, joka on normaalisti Adapteri-luokan viimeinen tehtävä. Muuhun koodiin tämä ei vaikuta: booleanilla on oletusarvon true, jota itse koodi käyttää, vain testit kutsuvat Adapteri-oliota false-arvolla.

Seuraavalla sivulla on ScalaTestinä toteutettujen testien nimet ja todistusaineisto sille, että kaikki 34 testiä menevät läpi:

```
Tests: succeeded 34, failed 0, canceled 0, ignored 0, pending 0
All tests passed.
```

- NTest (0,658 s)
  - TiedostonLukeminen.oikeellisuusTesti()
    - should find non-valid note names and lengths (0,020 s)
    - should find non-valid rest names and lengths (0,002 s)
    - should find valid note names and lengths (0,007 s)
    - should find valid rest names and lengths (0,002 s)
  - TiedostonLukeminen.lueTiedosto()__Recovering Badly Formulated Input
    - should #case1: too much white space (0,007 s)
    - should #case2: empty chords (0,002 s)
    - should #case3: chords written together without space (0,002 s)
    - should #case4: use of tabulator (0,002 s)
    - should #case5: not throw exception if input file is empty (0,001 s)
    - should #case6: not throw exception if input file has only #name-field but no note data (0,005 s)
  - TiedostonLukeminen.kasitteleKappaleenNimiJaTahtilaji()
    - should find time signature and title of the song when both hash-tagged in inputfile (0,010 s)
    - should use default values for Time signature and title of the song when neither hash-tagged in inputfile (0,0
  - TiedostonLukeminen.tarkistaVirheet() --as stub--
    - should find 3 syntax errors(= wrongly formulated note data) in file '3errors' (0,013 s)
  - NuottiPiirturi.lyricsBuffer
    - should contain right lyrics (0,066 s)
  - NuottiPiirturi.NuottiData and NuottiDataParitettu
    - should be of right length (0,021 s)
    - should their contents should have right lengths (0,026 s)
    - should their contents should have right note names (0,012 s)
  - NuottiPiirturi.tutkiEtumerkit()
    - should give out correct accidentals (0,028 s)
  - NeljasosaNuotti
    - should have correct attributes: nuppi, nimiMapissa, etumerkki and extraetumerkki (0,001 s)
  - KahdeksasosaPari.kuva
    - should draw eigth note couple stems down and ignore second flat (0,002 s)
  - PisteellinenNeljasosaNuotti.kuva
    - should draw dotted quarter note stem up (0,001 s)
  - Viivasto.kappale
    - should have song title and 2 staffs (0,012 s)
  - Viivasto.kasitteleLyriikat()
    - should cope with more lyrics than notes (0,001 s)
    - should cope with less lyrics than notes (0,001 s)
    - should cope with more letters in syllables than print area (0,004 s)
  - UI.loytyykoInputHakemistosta()
    - should give false when file not found in directory (0,013 s)
    - should give true when file found in directory (0,001 s)
  - UI.kayttajaValitseeMIDIPatchin() --as stub--___accept only key presses 1-7 and ENTER
    - should #case1: 2 non-acceptable, 1 acceptable value (0,000 s)
    - should #case2: only ENTER (0,001 s)
    - should #case3: only non-acceptable values (0,001 s)
  - simpleMIDIPlayerAdapter.muunnaMIDInuoteiksi()
    - should transform Nuottipiirturi.nuottiData to correct Buffer[(Buffer[Int], Double)] (0,045 s)
  - The program (integration test)
    - should produce song 'Flight of the Bumble Bee' similarily as in correctly printed file 'bumble.txt' (0,229 s)
    - should produce test file 'allViivastolleLaitettavaClasses' similarily as in correctly printed file 'all.txt' (0,031 s)
    - should produce test file '_accidentals' similarily as in correctly printed file 'accidentals.txt' (0,025 s)

8. Ohjelman tunnetut puutteet ja viat

Sinänsä ohjelman pitäisi toimia: en keksi miten sen saisi kaatumaan. Ja mikä tärkeintä nuotit printtautuvat, soivat ja tallentuvat tiedostoon oikein. Pieni fiba: skrollaus ei toimi oikein tapauksissa, joissa tahtiviivalle ei löydy paikkaa. Ja tietysti jos käyttäjä tekee vastoin ohjeita, esimerkiksi sointuja, joiden äänet ovat eripitkiä keskenään, ei tahtiviivan paikka eikä tästä johtuen etumerkkilogiikkakaan mene oikein.

Puutteet ovatkin toimintoja, joita voisi ohjelmaan lisätä:

1) **tahtiviivan yli sitominen**. Vaikutti työläältä ja mietitytti miltä sidontakaaret näyttäisivät tällä grafiikan granulariteetilla. Epäilys: todella rumilta ja pahimmillaan voisivat aiheuttaa sekaannusta käyttäjän yrittäessä erottaa toisistaan sidontaviivoja ja viivaston ja nuottien viivoja.

2) **kuudestoistaosien implementointi.** Tässä ainoa ongelma on ollut, että käyttämäni syöteformaatti on “täydellinen” nykyisellään: ⅛-nuotit ilmaistaan näin: “c1”, eli ei pituusdataa, koska se on lyhyin, mitä ohjelma hyväksyy. Miten 1/16-nuotit olisi pitänyt ilmaista “c1=”, kenties? “=” voisi viitata siihen että 1/16-nuotissa on 2 palkkia. Mutta sitten mietin, että syötedata on jo nyt melkoisen kompleksinen erimittaisine nuotteineen, sointuineen, etumerkkien merkitsemisineen, lyriikoineen ja tunnisteineen: meneekö keskivertokäyttäjän muisti-/tajuamiskapasiteetin yli pitää mielessä paljon detaljeja siitä, miten syöte tulee muotoilla. Ilman käyttäjätestausta epäilisin, että kyllä menee. Keep it simple!

Testasin hieman, minkä työmäärän olisi vaatinut 3) **kahdeksasosasointujen laittaminen samaan palkkiin.** Kuulostaa helpolta, mutta olisi vaatinut tällä arkkitehtuurilla kohtuutonta määrää koodia. Katso kuva sivulla 1: kaksi ensimmäistä kahdeksasosaa eivät nykyisin ole samassa palkissa, koska ensimmäinen on luokkaa *Sointu* ja toinen *KahdeksasosaNuotti.* Jokainen tapaus olisi pitänyt koodata erikseen: a) yksittäinen ⅛-nuotti liittyy ⅛-sointuun, b) samat toisin päin, c) kaksi ⅛-sointua peräkkäin. Yksittäisten nuottien/sointujen väliin ei voi pelkästään piirtää palkkia, koska nuottien varret voivat yksittäistapauksina osoittaa eri suuntiin, mutta parina niiden pitää mennä samaan suuntaan. Uskoisin että piirtämisen virhealttius olisi moninkertaistunut, ja toisaalta mikä on hyöty: vain toinen tapa merkitä peräkkäisiä kahdeksasosia, kun nykyinenkin toimii (ei vain ole niin hienon näköinen kuin oikeassa nuotinnoksessa käytetty versio).

Nyt jälkiviisaana olisi voinut olla “järkevämpää” luopua eri suuntaan menevistä varsista, niin kolmoskohdan koodi olisi ainakin lyhentynyt. Toisaalta olen koulutukseltani musiikin maisteri ja en mielelläni tee nuottien varsia vääriin suuntiin edes Scalalla.

9. Kolme parasta ja kolme heikointa kohtaa

<u>3 parasta:</u>
* nuottien ja äänen tuottaminen samasta käyttäjän syötetiedostosta ja samasta ohjelman sisäisestä tietorakenteesta Buffer[ViivastolleLaitettava]
* nuotit piirtyvät siististi, graafisesti kiva katsella
* edistyksellinen syötedatan virheidenkorjaussysteemi, jonka seurauksena ohjelma ei kaadu

<u>3 heikointa:</u>
* ei voi tehdä 1/16-osia eikä sitä vaikeampia rytmikuvioita
* ei tahtiviivan yli meneviä nuotteja

* ei toimi itsenäisenä exenä Eclipsen ulkopuolella (tosin tämä ei kuulunut kurssin vaatimuksiinkaan)

10. Poikkeamat suunnitelmasta

Tein kaiken mitä suunnittelin. Lisäksi laajensin syötteiden virheidensietokykyä, toteutin nuottidatan soittamisen useilla soundeilla, nuottien oikea-aikaisen skrollaamisen ja testasin suunniteltua laajemmin.

11. Toteutunut työjärjestys ja aikataulu

Nuottien piirtäminen aloitettiin suunnitellusti helpoimmista, eli ensin tehtiin ja testattiin perimisjärjestyksessä ensin olevaa luokkaa ennenkuin kannatti tehdä sen mahdolliset virheet perivä luokka. Aluksi jätettiin myös etumerkit pois ja kaikki nuotit tehtiin ennen sointuja.

Ensin tehtiin ohjelman runko, joka toimii oikeilla syötteillä oikein. Ei välitetty poikkeuksista, jos vahingossa syötetiedostossa oli väärin muotoiltua dataa. Sitten ruvettiin lisäämään logiikka, joka kierrättää nuotit virheiden tarkistus- ja oikaisumetodien kautta. Kun tämä toimi yksittäistapauksissa, suunniteltiin luuppi, jossa ohjelmaa ei tarvitse välillä sulkea, jotta syötetiedostoa voi editoida.

Yksikkötestaaminen aloitettiin vasta maaliskuussa. Tällöin koodi jo toimi 98% tapauksista. Tämän jälkeen tehdyt muutokset ovat lähinnä arkkitehtuurisia: luotiin uusi luokka UI, johon pystyttiin ulkoistamaan lähes kaikki käyttäjäinteraktio. Tästä oli 3 hyötyä: ohjelman rakenne selkiytyi ja testattaessa UI-toiminnot pystyttiin ohittamaan, jos oli tarkoitus testata esim pelkkää NuottiPiirturi-luokkaa ilman että käyttäjältä kysytään tiedoston nimeä. Lisäksi koodimäärältään paisunut TiedostonLukeminen-luokka pieneni.

Aikataulu: tuntimäärää ei ole laskettu, mutta aikaa kului huomattavasti enemmän kuin suunnittelin. Vaikka olenkin työelämässä, niin vapaa-aikaa tuntui kuitenkin olevan (koodaaminen iltakymmenestä yöyhteen tuli lähes jokailtaiseksi tavaksi), viikonloppuisin koodailin ja hiihtolomalla ehdin koodata todella paljon. GitHub-commiteista (214 kpl) https://github.com/niinasaarelainen/NuottiPiirturi voi nähdä että ohjelmaa/ yksikkötestausta on työstetty 59 päivänä. Tyypillinen koodauspäivä oli 2-5 tuntia, pari 9 tunnin päivääkin mahtui joukkooon.

Lisäksi aikaa on mennyt UML-kaavion tekoon ehkä 4 tuntia, tämän dokumentin kirjoittamiseen noin 30 tuntia, edellisten noin 20, GitHubin käytön opetteluun noin 3 tuntia ja ScalaTest-ympäristön rakentamiseen noin 10 tuntia (todella paljon vaikeuksia oli saada Eclipsen Run-toiminto pelittämään, eli testi-.class -tiedoston muodostaminen ei onnistunut vain ohjeita noudattamalla).

Lisäksi projektia koodattiin ennen ensimmäisen GitHub-commitin tekoa noin 40 tuntia. Tällöin koodailin ilman kokonaisnäkemystä ensimmäisellä mieleeni juolahtaneella algoritmilla: * luodaan lopullisen printtausrivin mittainen tyhjä viivasto Array[Array[Char]]:na * piirretään yksi nuotti kerrallaan syötepuskurista char-taulukkoon korvaamalla merkkejä *muuttujaa x (x-akseli) kasvatetaan nuottien viemän tilan mukaan * piirretään tahtiviiva, kun on sen vuoro. Jne. Tässä tuli huomattavia ongelmia x-muuttujan arvossa tehtäessä kahdeksasosapareja, lopputuloksen siisteyden kanssa ja sen kanssa, että piti etukäteen arvioida kuinka pitkä viivasto luodaan. Mahdotonta. Onneksi yliopistolehtori (ja mieheni) Matti Luukkainen sanoi: “Ei näin. Mietitäänpä vähän tietorakenteita.” Hyvä! Ainoastaan järkevällä nuottidatan tallentamisella (Buffer[ViivastolleLaitettava]) oli mahdollista soittaa kappale ja

saada siisti lopputulos, joka ei voi kaatua minkään mittaisilla nuottikombinaatioilla, koska viivaston pituus ei ole etukäteen lyöty lukkoon.

12. Arvio lopputuloksesta

Kiitettävä tai erinomainen ollakseen kolmessa kuukaudessa koodattu. Eniten pidän siitä, että sain muusikkolähtoisen syötedatan toimimaan. Uskoisin että muusikoiden on helppo muistaa nuottien merkintätavat, koska kaikki ovat musiikinteoriasta lähtöisin olevia asioita: nuotin pituus on juuri se mihin on totuttu: “kahden mittaiseen nuottiin laitan 2 viivaa”, eikä tarvitse ruveta laskelmoimaan hankalasti esimerkiksi nuotin alkamisaikaa murtolukuna 4/32, kuten projektisivulla. Tahtinumeroiden puuttuminen tekee toistoista erittäin helppoa. Esimerkiksi jos kappaleessa on 2 säkeistöä, saa toisen säkeistön nuotit copy-pastaamalla sama data sellaisenaan syötetiedostoon.

Lisäksi soitto-ominaisuus on mukava ja tarpeellinen lisä.

Enemmänkin ominaisuuksia olisi voinut laittaa, mutta ohjelmaan käytetty aika ei olisi ollut enää missään järkevässä mittasuhteessa kurssista saataviin opintopisteisiin.

Ohjelmarunko on helposti laajennettava: **sointumerkit**, jotka ohjelma voisi helposti soittaa (tiedän miten parilla rivillä koodataan minkä tahansa duurin tai mollin soittaminen). Tällöin voisi kuunnella miten melodia sopii mielessä soivaan sointuun ja sointumerkeistä nuottikuvassa olisi hyötyä mm. kitaristeille ja pianisteille. **GUI** ei pitäisi olla kohtuuttoman työn takana, koska tieto nuottien ominaisuuksista ja sijainneista on luokan Map-tietorakenteessa ja kentissä, sen kysyminen on jo toteutettu, ainoa ero olisi tiedon esittäminen eri grafiikalla. Nuotit **F-avaimella** olisi helppo tehdä muuttamalla Map[“nuotin nimi” → Int]:n arvoja ja laittamalla Viivasto-olion konstruktori piirtämään G-avaimen sijaan F-avaimen kuva.

13. Viitteet

Kirjallisuus:

Scala for the Impatient (Cay S. Horstmann)

Internet-linkit:

Scala Standanrd Library: http://www.scala-lang.org/api/current/
Pareja ja hakuja: https://plus.cs.hut.fi/o1/2016/k09/osa01/
Tiedostojen käsittely: https://plus.cs.hut.fi/o1/2016/k09/osa03/
Tuple: https://plus.cs.hut.fi/o1/2016/k09/osa04/
Periytyminen: https://plus.cs.hut.fi/o1/2016/k06/osa04/
MIDI-Synthesizer: http://docs.oracle.com/javase/tutorial/sound/MIDI-synth.html ja https://docs.oracle.com/javase/8/docs/api/javax/sound/midi/MidiSystem.html
Yksikkötestaus: http://www.scalatest.org/user_guide, https://plus.cs.hut.fi/studio_2/2017/k16/osa02/, https://plus.cs.hut.fi/studio_2/2017/k16/osa03/
UML-kaavion teko: https://yuml.me/, https://yuml.me/diagram/scruffy/class/samples
Versionhallinta: https://github.com/niinasaarelainen/NuottiPiirturi
https://github.com/mluukkai/OTM2016/wiki/Viikon-3-paikanpaalla-tehtavat
Single responsibility -periaate https://en.wikipedia.org/wiki/Single_responsibility_principle

Executable jar: https://plus.cs.hut.fi/o1/2016/k05/osa02/#lisamateriaalia,
http://alvinalexander.com/scala/sbt-how-to-compile-run-package-scala-project
(yritettiin tehdä executable jar, mutta ei pienellä vaivalla onnistuttu, teen myöhemmin…)

14. Liitteet

Lähdekoodi, yleissuunnitelma ja tekninen suunnitelma Eclipse-projektin sisällä.

Liite 1: Kolme ajoesimerkkiä ohjelman käytöstä

ajoesimerkki #1: käyttäjä valitsee tiedoston listauksesta, katsoo ja kuuntelee sen soundilla nro 4, tallettaa nimellä Finlandia.txt ja lopettaa

> Finlandia, ENTER
> 4, ENTER *(kuunnellaan soundilla nro 4)*
> 0, ENTER *(ei kuunnella uudestaan)*
> Finlandia, ENTER *(tallettuu output-kansioon nimellä Finlandia.txt)*
> ENTER *(lopetus)*

ajoesimerkki #2: käyttäjä tekee oman tiedoston, jossa on virheitä

Ohjelman ulkopuolella on tehty tiedosto nimeltä omabiisi1.txt ja talletettu hakemistoon input.
Tiedoston sisältö:

```
z c1  <c1,e2,g11><c1,e2,g1>
c1  h2f1 g1-
```

> omabiisi1.txt, ENTER
syöte 'g11' on virheellinen: väärä formaatti. Muistathan syntaksin: esim. alennettu e on Eb, ei es
Virhe on rivillä 1
Korjaa äsken valitsemaasi tiedostoon ja paina ENTER, kun tiedosto on tallennettu.
> ENTER *(ensin on korjattu ohjelman huomaama virhe)*
syöte 'h2f1' on virheellinen: liian pitkä nuotin nimi
Virhe on rivillä 2
Korjaa äsken valitsemaasi tiedostoon ja paina ENTER, kun tiedosto on tallennettu.
> ENTER *(ensin on korjattu ohjelman huomaama virhe)*
> ENTER *(ei haluta kuunnella, printtautuu koko biisi kerralla)*
> ENTER *(ei haluta tallettaa)*
> ENTER *(lopetetaan)*

ajoesimerkki #3: usean toiminnon sarja

> FürElise, ENTER
> 1, ENTER *(kuunnellaan soundilla nro 1, joka on piano)*
> ENTER *(kuunnellaan uudestaan)*
> 0, ENTER *(ei kuunnella uudestaan)*
> Elise *(tallennetaan)*
> 1, ENTER *(ohjeet)*
> 2, ENTER *(listaa kappaleet ja nuotinna -toiminto)*

> SMOKE, ENTER *(isoilla ja pienillä kirjaimilla ei ole väliä kappaleen nimessä)*
> 6, ENTER *(kuunnellaan soundilla nro 6, aka rokkibändi)*
> 0, ENTER *(ei kuunnella uudestaan)*
> ENTER *(ei haluta tallettaa)*
> ENTER *(lopetetaan)*

Liite 2: Alla olevassa linkissä on nuottiesimerkki, jossa on esitelty kaikki eripituiset nuotit ja tauot. Joka aika-arvosta löytyy varsi ylös ja alas. Myös soinnuissa näkyvät kaikki aika-arvot kokonuotista kahdeksasosanuottiin. Viimeinen nuottirivi toimii myös esimerkkinä, mitä ohjelma tekee jos tahtiviivalle ei löydy paikkaa: se jätetään piirtämättä, mutta seuraava kohta joka on jaollinen tahtilajilla tuottaa viivan.

https://github.com/niinasaarelainen/NuottiPiirturi/blob/master/output/all.txt

Alla vielä pisin syötetiedosto (631 riviä). Kappaleessa on myös “sanoitus”, jonka funktiona on testata että liian pitkät tavut (sanat tarkoituksella jätetty tavuttamatta) eivät kaada ohjelmaa, ja pysyvät synkroonissa nuottien kanssa.

https://github.com/niinasaarelainen/NuottiPiirturi/blob/master/output/bumble.txt