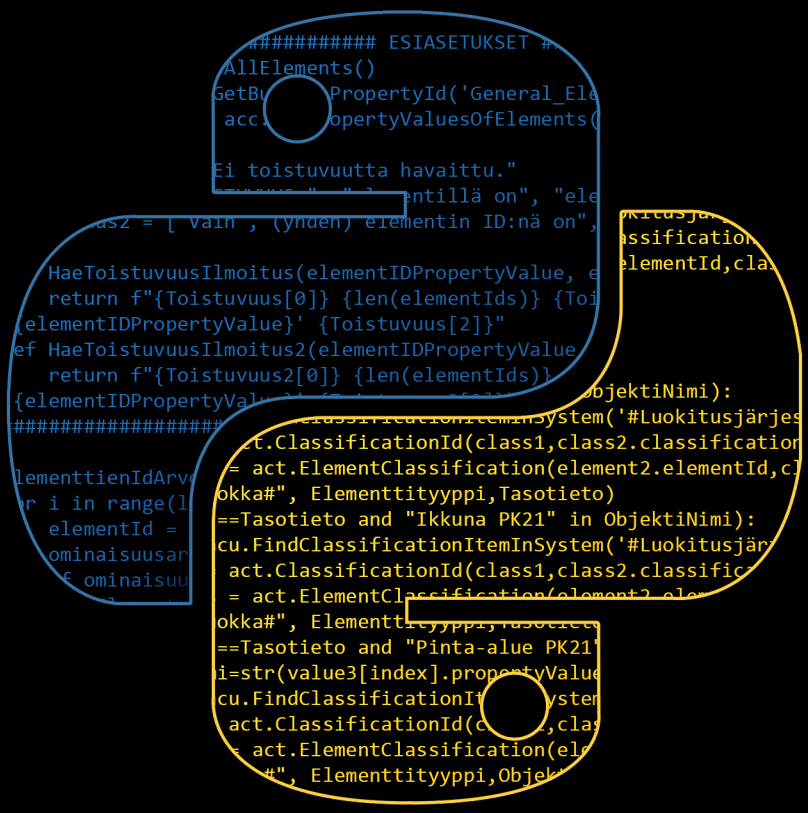
Automatisoi se!

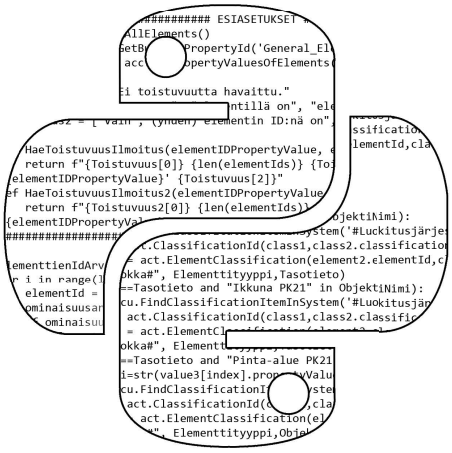
Ohjelmointi osana arkkitehtisuunnittelua



Diplomityö
Oulun yliopisto
Teknillinen tiedekunta
Arkkitehtuurin yksikkö
Juho Tiltti
Kevät 2022

Automatisoi se!

Ohjelmoinnin merkitys arkkitehtisuunnittelussa



OULUN YLIOPISTO

Arkkitehtuurin maisteriohjelma
Diplomityö
Juho Tiltti
Kevät 2022

TIIVISTELMÄ

Tekijä(t): Juho Tiltti
Diplomityön nimi: Automatisoi se! Ohjelmoinnin merkitys arkkitehtisuunnittelussa
Työn pääohjaaja(t): Aulikki Herneoja
Työn valmistumislukukausi ja -vuosi: Kevät 2022
Sivumäärä: 80

Arkkitehdin ammatissa ohjelmointiosaaminen ei tule ensimmäisenä mieleen, kun mietitään ammatin harjoittamisen kannalta välttämättömiä osattavia kokonaisuuksia. Kuitenkin jo tänä päivänä ohjelmointitaitoa vaaditaan esimerkiksi algoritmisessa suunnittelussa, vaikka sitä ei heti näkisikään aiheeksi, jossa ohjelmointiosaaminen jollain tasolla on lähes välttämätöntä.

Diplomityössäni tutkin kuinka ohjelmoinnin hallitseminen sekä käytetyn ohjelmointikielen rakenteen ymmärtäminen toimisivat arkkitehtisuunnittelun apuvälineenä. Tavoitteenani oli löytää konkreettisia kohteita, joissa pystyisin hyödyntämään itse kirjoitettuja ohjelmia päivittäisissä työtehtävissä sekä osoittamaan valmiiden ohjelmien avulla ohjelmointitaidon tuomat hyödyt sekä tarpeellisuus osana arkkitehdin työkalupakkia.

Työni on luonteeltaan tutkielmapainotteinen ja se jakautuu teoria- sekä toteutusosioihin. Aloitin aineiston kokoamisen tutustumalla aiheesta löytyvään kirjalliseen materiaaliin. Syvennyin kokoamastani aineistosta varsinaisesti osioihin, jotka käsittelivät ohjelmointia yleisellä tasolla, siihen liittyvää termistöä sekä tämän päivän ilmiöitä ohjelmoinnin ja arkkitehtuurin yhteensovittamisesta.

Toteutusvaiheessa tein varsinaisen koodaamisen käyttäen Python-ohjelmointikieltä. Kirjoitin ohjelmat erilaisia työprojekteja varten projekteissa ilmenneiden automatisointitarpeiden perusteella. Koodaamisen pohjana käytin teoriavaiheessa opiskelemaani kirjallista materiaalia sekä Python-ohjelmointiin liittyneitä harjoitustehtäviä. Lopulliset ohjelmat rakensin eri vaiheissa kirjoitettujen sekä testattujen pienempien ohjelmien pohjalta. Lopuksi ajoin valmiit ohjelmat Archicad-projekteihin.

Toteutusvaiheen aikana kirjoitin useamman käyttökelpoisen ohjelman, joiden joukosta valitsin kolme esiteltäväksi tässä diplomityössä. Nämä kolme valikoituivat niiden toiminnan perusteella. Jokainen ohjelma vastaa eri tarpeeseen, vaikka rakenteeltaan ne ovat lähellä toisiaan. Näin minun oli mahdollista osoittaa, että Python-kieli on tarpeeksi monipuolinen kuitenkin ollen samalla yksinkertainen, jotta ohjelmoinnin hyödyntäminen arkkitehtisuunnittelussa olisi perusteltua.

Asiasanat: Ohjelmointi, Python, Automatisointi, Tietomalli, Arkkitehtisuunnittelu

ABSTRACT

Author(s): Juho Tiltti
Title of the diploma thesis: Automate it! The importance of programming in architectural design
Supervisor(s): Aulikki Herneoja
Term and year when the thesis was submitted: Spring 2022
Pages: 80

In the profession of an architect, programming skills do not come to mind first when thinking about the competencies that are necessary for practicing the profession. However, programming skills are required, for example, in algorithmic design, even if it is not immediately considered a subject requiring programming skills.

In my diploma thesis, I study how mastering programming and understanding the structure of the used programming language would serve as an aid to architectural design. The aim was to find concrete targets for using self-written programs in day-to-day tasks and to demonstrate the benefits and necessity of programming skills as part of an architect's toolkit.

My work is dissertation-oriented in nature and is divided into theoretical and implementation sections. I started compiling the material by looking first at the written material on the subject. From the material I collected, I delved into the parts that dealt with programming in general, the related terminology, and today's phenomena about the coordination of programming and architecture.

In the implementation phase, I did the actual coding using the Python language. I wrote the programs for different work projects based on the automation needs that emerged in the projects. As a basis for coding, I used the written material that I studied in the theoretical phase as well as the exercises related to Python programming. I built the final programs based on the smaller programs which were written and tested at different stages. Finally, I ran ready-made programs for Archicad projects.

During the implementation phase, I selected from several useful programs three to be presented in my diploma thesis. These three were selected based on their performance. Each program was written for a different need, but structure in those programs was similar compared to each other. In this way, I could demonstrate that the Python-language is diverse enough, yet simple to justify the use of programming in architectural design.

Keywords: Programming, Python, Automation, Building information model, Architectural design

SISÄLLYSLUETTELO

1 JOHDANTO

Melkein jokainen ammatti vaatii tänä päivänä tietotekniikan hallitsemista. Joissain tehtävissä sitä tarvitaan vähemmän, toisissa enemmän. Joihinkin ammatteihin tietotekniikka tekee taas uudenlaista läpimurtoa käytettyjen työskentelytapojen suhteen, ja mahdollistaa vanhojen rutiinien täysin uudenlaisen tarkastelun sekä työtapojen kehittämisen. Arkkitehdin ammatin voidaan katsoa kuuluvan ryhmään, jossa eteenpäin kehittynyt tietotekniikka ja sovellusmaailma muovaa jälleen suunnitelmien toteutuksen vakiintuneita käytäntöjä.

Nykypäivänä perinteisen kynän rinnalla ovat olleet jo useita vuosia eri ohjelmistoyritysten tuottamat CAD-ohjelmistokokonaisuudet, joiden avulla arkkitehtien, rakennus- ja muiden rakennustekniikkaan liittyvien suunnittelijoiden ajatukset toteutettavista ratkaisuista siirtyvät näkyviksi ohjeiksi ja suuntaviivoiksi suunnitelmien toteuttajalle. Välttämättä pelkät CAD-sovellukset eivät kuitenkaan riitä tuottamaan helposti useita vaihtoehtoisia ratkaisuja ilman suurta työmäärää. Tällöin CAD-ohjelmat tarvitsevat mahdollisesti rinnalleen ohjelmiston, joka hyödyntää algoritmeja sekä algoritmien sisällä olevia muutettavia, tai ennalta määrättyjä parametreja. Seuraavassa vaiheessa tieto siirretään linkitysten kautta haluttuun CAD-ohjelmaan. Parametrisen suunnittelun tehokkuus perustuu siihen, että erilaisia ratkaisuja voidaan tuottaa lähes rajattomasti parametreja sisältävien algoritmien avulla automaattisesti pelkästään parametriarvoja muuttelemalla.

Suunnittelutyössä tietomalli on yksi työväline muiden joukossa, jonka merkitys korostuu päivä päivältä enemmän. Tietomalli on kirjaimellisesti malli, jossa elementit sisältävät tarkkaa dataa esimerkiksi toteutustavasta, valmistajasta, u-arvoista, lukitustyypeistä, hoito-ohjeista ja niin edelleen. Voidaankin siis sanoa, että pelkkä 3D-massamalli ei vielä täytä sisällöllisesti tietomalli-nimikkeen vaatimuksia vaan mallissa olevien elementtien todellakin pitää sisältää tietoa pelkän visuaalisen informaation lisäksi. Parametrisen suunnittelun apuna käytetyt ohjelmistot, esimerkiksi Rhinoceros 3D ja linkkiohjelmisto Grasshopper eivät taivu juuri tähän tarkoitukseen.

Tietomallin vaatiman datan syöttäminen tapahtuu yleensä käsin, Excel-taulukkojen avulla, tai kopioimalla tarvittavan tiedon sisältämää elementtiä projektin sisällä. Tässä mekaanisesti syötetyssä tai kopioidussa datassa piilee kuitenkin ristiriitaisuuksien mahdollisuus. Ne voivat hyvin olla esimerkiksi kirjoitusvirheitä tai väärän tiedon syöttäminen väärään elementtiin.

Yhtenä tapana tiedon syöttämiseen ja tarkastamiseen voidaan käyttää sovelluksista tallennettavia taulukoita. Arkkitehtisuunnitteluun soveltuvissa suunnittelusovelluksissa, kuten Revitissä ja Archicadissa, datan syöttämisen ja tarkistuksen automatisointiin on kuitenkin ratkaisu, joka on Python ohjelmointikieli. Archicad mahdollisti Python-linkin kokeellisena ominaisuutena sovelluksen käyttäjille vuonna 2020 ja Revit hieman ennen tätä. Pythonin vahvuus piilee sen avoimuudessa. Ohjelmointikielellä on laaja yhteisö, josta voi saada tarvittaessa apua sekä ohjeita omien sovellusten kirjoittamiseen.

Tässä diplomityössä tavoitteenani on tarkastella kuinka ohjelmoinnin hallitseminen sekä sen ymmärtäminen toimivat arkkitehtisuunnittelun apuvälineenä. Jos ohjelmoinnista ei ole aikaisempaa kokemusta, niin tällöin järjestys opettelun suhteen on juuri edellisessä lauseessa mainitun järjestyksen mukainen. Jotta voidaan ymmärtää käyttömahdollisuudet, pohjalle täytyy rakentaa kielen perusteiden hallitseminen sen tarjoamien mahdollisuuksien saavuttamiseksi. Varsinaisina työvälineinäni toimivat ohjelmointikieli Python, IDE Microsoft Visual Studio sekä Archicad-ohjelmisto. Erityistä huomiota kiinnitän siihen, kuinka paljon itse kielen opettelu vaatii aikaa. Ennen tämän diplomityön aloitusta itselläni oli vain satunnaista kokemusta ohjelmoinnista yleensä. Oma taustani antaa näin ollen hieman peilauspintaa siihen, minkälaista ajallista panostusta ohjelmoinnin opettelu vaatii ennen kuin varsinaisia tuloksia on nähtävissä.

Työni on luonteeltaan tutkielmapainotteinen. Se koostuu teoreettisesta taustatyö- ja konkreettisemmasta toteutusvaiheesta sekä tulosten käsittelystä. Teoriaosiossa syvennyn varsinaisesti ohjelmoinnin käsittelemiseen, sen termistöön sekä tarkastelen tämän päivän ilmiöitä ohjelmoinnin ja arkkitehtuurin yhteensovittamisesta. Teoreettinen osuus

toimii näin myös osana oman ohjelmointiosaamisen kehittämistä ja pohjustaa vahvasti varsinaisen toteutusvaiheen vaatimaa ymmärrystä ohjelmointikielen toimintalogiikasta.

Toteutusvaiheessa teen varsinaisen koodin kirjoittamisen Python-kielellä. Python-tiedostot kirjoitan erilaisia työprojekteja varten niissä ilmenneiden automatisointitarpeiden mukaan. Samalla tarkastelen voisiko jo ennestään tehtyjä .py-tiedostoja käyttää tehokkaasti pienillä muutoksilla tai sellaisenaan muissakin projekteissa. Pyrin luomaan selkeitä ja helposti lähestyttäviä koodikokonaisuuksia, jotta niiden käyttötarkoituksen ymmärtäminen sekä muokkaaminen olisi helppoa myös heille, joilla ei ole ohjelmoinnista aikaisempaa kokemusta.

Algoritmi=	*Yksityiskohtainen kuvaus tai ohje järjestyksestä, jolla toiminnot voidaan suorittaa työstettävän ongelman ratkaisemiseksi sekä halutun lopputuloksen saavuttamiseksi.*
API=	*Application programming interface eli ohjelmointirajapinta on määritelmä, jonka mukaan ohjelmat voivat keskustella keskenään tietopyyntöjen sekä tietojen vaihdon kautta.*
.bat-tiedosto=	*.bat(Batch)-tiedosto on tekstitiedosto, joka sisältää komentokielijonon. Komentojono sisältää tavallisia DOS-komentoja ja ne toimivat myös Windows komentorivin kautta. Batch-tiedostoja voidaan käyttää ohjelmien käynnistämiseen sekä ajamiseen.*
BIM-malli=	*Builgind Information Model eli rakennuksen tietomalli*
Debuggeri=	*Ohjelmiston ohjelmointivirheiden jäljitykseen käytettävä työkalu. Debuggerit ovat nykyään osana ohjelmointiympäristöjä. Virheenjäljitys osoittaa tarkan pysähdyspisteen/-pisteet (breakpoint), joissa koodissa olevat virheet ilmenevät.*
Formaali kieli=	*Äärellisen pituinen merkkijonojen joukko, jotka muodostuvat äärellisestä aakkostosta. Itse kieli voi kuitenkin sisältää äärettömän määrän merkkijonoja eli stringejä.*
Funktio=	*Aliohjelma, joka on ohjelmoinnissa itsenäinen ohjelman osa. Se suorittaa tietyn toiminnon, jota voidaan kutsua eri puolilta pääohjelmaa tai muista aliohjelmista.*
Garbagecollection=	*Automaattinen roskienkeräys. Muistinhallinta järjestelmä, jossa roskienkerääjä pyrkii poistamaan automaattisesti tiedot muistista, joihin sovellus ei tule enää viittaamaan ja vapauttamaan niiden käyttämän muistitilan uudelleen käytettäväksi.*

IDE=	*Integrated development environment eli ohjelmointiympäristö. Ohjelmointiympäristöä käytetään ohjelmistojen suunnitteluun ja toteutukseen. Periaatteessa mitä tahansa tekstieditoria ja ohjelmistokielen kääntäjää voidaan käyttää IDE:nä. Editorin avulla kirjoitetaan lähdekoodi, joka käännetään komentoriviltä käytettävältä kääntäjällä.*
IFC=	*Industry Foundation Classes. IFC on kansainvälinen rakennusalan ISO/PAS 16739 standardi oliopohjaisen tiedon siirtoon eri ohjelmistojen välillä.*
LISP=	*List Processing (Programming Language). Se on toiseksi vanhin ohjelmointikieli, joka on jakautunut useisiin murteisiin.*
Muuttuja=	*Symbolinen tietovarasto, josta tietoa voidaan hakea ja johon voidaan tietoa kirjoittaa, jos muuttuja ei ole sisällöltään ennalta määritetty eli vakio.*
Ohjelma =	*Sarjaan järjestetty joukko ennalta kirjoitettuja komentoja, käskyjä, tai suunnitelma, tai proseduuri, jota seuraten tietokone suorittaa sille valmistellun tehtävän kysytyn ratkaisun saavuttamiseksi.*
Ohjelmisto=	*Yhdestä tai useammasta tietokoneohjelmasta, väliohjelmasta, ohjelmistokehyksestä tai ohjelmakomponentista koostuva kokonaisuus.*
Ohjelmointi=	*Komentokäskyjen kirjoittaminen jollakin tavalla. Tyypillisesti käskyt kirjoitetaan formaalilla kielellä.*
Parametri=	*Funktiolle tai käskylle välitettävä tieto. Aliohjelmassa parametrit voivat olla muuttujia tai viittauksia muuttujiin.*
Primitiiviset toiminnot=	*Tietotyypit, jotka ovat oletuksena määriteltyjä ohjelmointikielessä. Näitä ovat esimerkiksi kokonaisluku, kelluvat datatyypit sekä merkistöt.*

PyPi=	*Python Package index on Pythonin virallinen kolmannen osapuolen pakettivarasto Python-kehittäjille. Sivuston sisältö avointa dataa ja käytettävissä vapaasti. Sivustolta ladattavat paketit sisältävät tarvittavat kirjastotiedot, joita vaaditaan .py-päätteisten tiedostojen ajamiseen halutussa ohjelmistossa.*
Python=	*Avoimen lähdekoodin monipuolinen, tulkattava ohjelmointikieli. Sitä voidaan käyttää esimerkiksi vaativassa ja tieteellisessä laskennassa. Pythonilla kirjoitetut ohjelmat voidaan ajaa sellaisenaan heti, kääntämättä niitä ensin.*
Renderointi=	*Malliin määritellyt pintamateriaalit leivotaan, eli upotetaan osaksi visuaalista kuvaa. Itse renderoinnissa materiaalin heijastukset, syvyyskartat lasketaan osaksi lopullista bittikarttagrafiikkaa.*
String=	*Merkkijono*
Sovellusohjelma=	*Tietokoneohjelma, joka on suunniteltu helpottamaan tietyn tehtävän tai ongelman ratkaisua.*
Visual Studio Code=	*Ohjelmointiympäristö (ks. IDE)*
VPL=	*Visual Programming Language eli visuaalinen ohjelmointikieli. Sillä tarkoitetaan kaikkia ohjelmointikieliä, joissa koodit kirjoitetaan graafisesti käyttämällä sekä yhdistämällä toimintokomponentteja.*

2 UUDEN OPPIMISTA

Ohjelmointi, merkkijonot, komennot. Näemme edellä mainittuja käsitteitä joka puolella ympäristössämme. Osa on luonnon omia DNAssa ja RNAssa esiintyviä merkkijonoja, jotka rakentuvat miljoonien vuosien kehityksen tuloksena komentojen perusteella ennalta määritellyiksi ketjuiksi. Osa syntyy taas ohjelmistokehittäjän aivoissa signaaleina sormille, jotka purkavat nämä signaalit tekstimuotoon pitkiksi ja useita tuhansia merkkejä sisältäviksi koodikokonaisuuksiksi.

Ohjelmistokehittäjän kirjoittamalle koodille on lähtökohtaisesti aina jokin tarve. Se voi olla kokonaan uusi ohjelma tiettyyn käyttötarkoitukseen, tai vanhan olemassa olevan sovelluksen ohjelmistopäivitystä varten tehtävä lisäys. Tarve voi vastaavasti myös olla itsensä kehittäminen sekä uuden oppiminen ja tätä kautta opitun hyödyntäminen jossain aivan muussa toimenkuvassa. Pieni ajallinen uhraus ohjelmointitaitojen kehittämiseen voi mahdollistaa suuren hyödyn pitkällä aikavälillä, ja vapauttaa resursseja työskentelyyn varsinaisten ratkaistavien suunnitteluongelmien parissa.

Ohjelmistokehitykseen liittyvät voimakkaasti termit ohjelmistoarkkitehtuuri sekä ammattinimike ohjelmistoarkkitehti. Ohjelmistoarkkitehti on laajan kokemuksen omaava asiantuntija, jonka tehtävänä on yleensä vastata uusien ohjelmistotuotteiden korkeatasoisesta suunnittelusta sekä uuden koodin suunnittelumenetelmistä. Ohjelmistoarkkitehtuurilla tarkoitetaan taas ohjelmistojen perusrakenteita, joista koko ohjelmisto muodostuu.

Tarkastelen diplomityössäni, miten ohjelmointi ja arkkitehtuuri kahtena erillisenä merkityksenä voisivat yhdistyä sekä tukevatko ne toisiaan. Haluan selvittää, minkälainen on ohjelmoiva arkkitehti, mitä hänen tekemänsä työ sekä ideat mahdollistavat arkkitehtuurin saralla, ja onko tietotaitoa mahdollista levittää kollegoille. Antaako vapaa ja yhteinen ohjelmointikieli mahdollisuuden isompaan evoluutioon, jossa pienet ohjelmat kehittyvät eri ihmisten täydentäminä varsinaisiksi ohjelmistoarkkitehtuurin sisältäviksi ohjelmiksi, joihin voi lennosta vapaasti lisätä omaa koodia. Ehkä se on lähempänä kuin huomaammekaan.

2.1 Rakentaminen, suunnittelu ja digitalisaatio

Yleisesti digitalisaatiolla tarkoitetaan tietotekniikan yleistymistä arkielämän jokapäiväisessä toiminnassa. Kännykät ovat tästä yksi hyvä esimerkki. Vielä kaksikymmentä vuotta sitten perinteinen lankapuhelin löytyi melkein jokaisesta kodista. Tänään matkapuhelimet ja langaton tiedonsiirto ovat syrjäyttäneet langallisen puhelimen äänidatan siirrossa, eivätkä operaattorit välttämättä enää edes tarjoa mahdollisuutta palata vanhaan.

Rakentamisen alalla digitalisaation tulo ja kehitys ovat parhaiten nähtävissä projektiin liittyvien tehtävien kautta. Tietokoneet ovat nopeuttaneet ja helpottaneet valtavasti monimutkaisten suunnitteluratkaisujen tarkastelua ja mahdollistaneet entistä tarkemman suunnittelutason. Rakennusprojektien aikataulutukset sekä niiden seuranta hoidetaan ja on hoidettu jo vuosia taulukko-ohjelmia käyttäen. Aikataulutaulukot voidaan taas kytkeä osaksi kustannuslaskentasovellusta ja kyseisiä laskelmia täydennetään rakennusosien kustannustiedoilla.

Suunnittelupuolella on käytössä suunnittelualoittain spesifioidut suunnitteluohjelmat, eli niin sanotut CAD-ohjelmat (*Computer-Aided Design*). Ensimmäiset CAD-pohjaiset suunnittelusovellukset aloittivat uuden aikakauden mahdollistamalla kynän ja paperin korvaamisen kokonaan digitaalisella suunnitteluympäristöllä. Viivat sekä piirustukset alkoivat rakentua sekä kehittymään tietokoneen näytölle, ja kymmenet erilaiset suunnitelmat voitiin tallentaa yhdelle kovalevylle sekä levittää sähköisesti suunnittelijalta toiselle.

Tietotekniikan kehityksen yhteydessä myös suunnitteluohjelmat ovat kehittyneet versio versiolta. Sovellukset ovat parantuneet ensin pelkkien viivojen piirtämisestä mahdollistamaan 3D-mallien rakentamisen (Burry, 1997), ja tämän jälkeen vähitellen myös varsinaisen tiedon syöttämisen itse malliin sekä 3D-mallin elementteihin. Kaksi viimeiseksi mainittua yhdessä muodostavat tietomallin. Siitä huolimatta, että tietomalleja on voitu tehdä ja tehty jo useita vuosia, niin vasta nyt

mallintamisen tarkkuustasot alkavat lähentyä pistettä, jossa vanhat piirustukset voidaan korvata rakennuslupaa haettaessa IFC-muotoon tallennetulla tietomallitiedostolla. Ensimmäinen hyväksytysti käsitelty tietomallipohjainen rakennuslupa käsiteltiin vuoden 2021 kesäkuussa Järvenpään rakennusvalvonnassa (Cloudpermit, 2021).

2.1.1 Tietomallien merkitys suunnittelussa

Mietittäessä tietomallin positiivisia aspekteja, voidaan todeta, että tiedon keskittäminen on sen yksi vahvimpia puolia (Jäväjä & Lehtoviita 2016, s. 30). Monissa projekteissa joudutaan suunnitteluvaiheessa vielä kuitenkin tukeutumaan useisiin eri sovelluksiin kokonaisuuden läpiviemiseksi, jolloin sama tieto voi esiintyä useassa eri paikassa toisistaan irrallisina palasina. Tiedon sijainnin pirstaleisuus sekä päällekkäisyys tarkoittaa sitä, että muutosten yhteydessä jokainen tiedosto, jossa muuttunutta tietoa voi olla, joudutaan tarkastamaan erikseen. Yhteen paikkaan syötetty tieto, josta se voidaan tarpeen vaatiessa myös linkittää muualle, vähentää päällekkäisyyksistä aiheutuvaa työtä sekä ristiriitaisuuksien mahdollisuutta.

Vaikka tietomallilla voidaan vähentää ristiriitaisuuksia, niin silti BIM-mallin toteutus voi aiheuttaa tunteen valtavasta lisätyöstä muun suunnittelutyön ohella. Kuitenkin tarkasteltaessa tilannetta tarkemmin, voidaan todeta, että projektien kokoluokan mukaan, vastaavia tietomääriä on vaadittu myös aikaisemmin. Tiedon sijainti on ollut tuolloin vain eri ja sen kirjaaminen on tehty omana työvaiheenaan. Tietomallin toteutuksessa tiedon syöttäminen sulautuu osaksi muuta työskentelyä mallin parissa. Varsinainen työmäärän kasvu johtuu ennemminkin ulkopuolelta tulevien paineiden takia, joita voivat olla esimerkiksi uudet määräykset tai tilaajan vaatimukset.

Tietomallit ovat olleet eri alojen suunnittelijoiden suunnitelmien yhteensovittamisen työkaluna tärkeitä ja tänä päivänä ne ovat lähes välttämättömiä (Jäväjä & Lehtoviita 2016, s. 26). Jos tekstimuotoista dataa löytyy elementeistä jo yhteensovitusvaiheessa, voidaan saada myös kiinni mahdolliset poikkeavuudet tai tarkentaa määrityksiä tilaajan tai määräysten mukaan. IFC-tiedosto ei ole kuitenkaan ainoa, jota voidaan

hyödyntää suunnittelun edetessä, tai arkistointia ajatellen. Esimerkiksi Autodesk Revit ja Graphisoft Archicad mahdollistavat tietomallin julkaisemisen myös BIM hypermodel eli hypermalli muodossa.

2.1.2 Tietomallien merkitys rakentamisessa

Vaikka tietomallit ovat vahvasti käytössä suunnittelupuolella, ne eivät ole saaneet tuulta alleen samalla tavalla rakennustyömaalla. Tänäkin päivänä suunnitelmat toimitetaan paperimuotoisena työmaalle ja siellä edelleen rakennustyöntekijän työmestalle. Paperituloste on selkeä ja yksinkertainen. Se toimii ilman sähkövirtaa ja havainnointi voidaan hoitaa nopeasti tulostettua asiakirjaa silmäillen. Digitaalimuotoinen suunnitelmapaketti taas vaatii tarkastelua varten soveltuvan laitteen, esimerkiksi henkilökohtaisen tablettitietokoneen jokaiselle työmaalla olevalle työntekijälle, käyttöönoton opastuksen sekä tarpeen vaatiessa tietenkin sähkövirtaa laitteen lataamista varten. Tästä näkökulmasta katsottuna alkupanostus on huomattavasti suurempi digitaalisella kuin fyysisellä suunnitteluasiakirjalla.

Kun asiaa mietitään kuitenkin laajalta skaalalta ja pidemmällä tähtäimellä, niin hyper- että IFC-mallin edut ja hyödyt alkavat nousta esiin. Suunnitelmien pyörittäminen työmaalla tablettitietokoneen kautta mahdollistaisi kattavan tarkastelun halutusta alueesta rakentamisteknisen toteuttamisen näkökulmasta. Digitaalinen välineistö mahdollistaisi välittömän kommentoinnin rakentajan ja suunnittelijan välillä. Haastavampien muotojen toteutuksen apuna 3D-malli on myös huomattavasti tarkoituksenmukaisempi kuin pelkkä 2D-tuloste (Jäväjä & Lehtoviita 2016, s. 56 – 57).

Aikataulua ajatellen suunnitelmat voitaisiin digitaalisessa muodossa saattaa rakennustyöntekijän käyttöön heti tarkennusten valmistumisen jälkeen. Paperitulosteita käytettäessä aikaa joudutaan varaamaan suunnitelmien taitolle suunnitteluohjelmistossa, suunnitelmien tulostamiseen kopiolaitoksella sekä tulosteiden toimitukseen työmaalle. Välikäsien poistuminen ketjusta sujuvoittaisi tiedon siirtämisen prosessia huomattavasti. Vaarana voi olla kuitenkin, että paperitulosteiden toimituksen pois jäämisen takia, säästynyt aika

heijastuu tulevaisuudessa suunnitteluaikataulujen tiukentumiseen. Tällöin digitoinnilla saavutettu ajallinen hyöty arkkitehtisuunnittelun osalta kuitenkin kumottaisiin, kun vapautuvaa aikaa ei voisikaan kohdentaa varsinaisten suunnitelmien laadukkaaseen toteuttamiseen. Toinen vaaran aiheuttaja voi olla suunnittelijoiden ja työmaan välinen kommunikointi tietomallia apuna käyttäen. Pahimmillaan kommunikoinnista voisi muodostua jatkuvalla tietotulvallaan suunnittelijan työn paha keskeyttäjä. Toisessa ääripäässä kommunikointi voisi taas puuttua kokonaan.

2.2 Tietomallien merkitys rakennuksen hallinnoimisessa

Luvussa 2.1.1 mainittu hypermalli ei sovellu sellaisenaan suunnittelijoiden yhteensovitukseen, vaan sen pääsääntöinen tarkoitus on toimia tilaajan tai rakentajan suuntaan tiedonvaihdon välineenä. Malli toimii pilvipalvelusta käsin, johon voi liittyä nettiselaimessa suunnittelijan jakaman linkin kautta, tai erillisen tallenteen kautta. Hypermalli sisältää suunnitteluohjelmassa määritellyt asiakirjat sekä dokumentit. Näitä voivat olla esimerkiksi 3D-malliin upotetut 2D-suunnitelmat, mallin pintakäsittelyt renderoituna ja elementteihin syötetyt tiedot. Tämä yksityiskohtaisen tietosisällön omaava lopullinen malli voitaisiin jo nykyään hyödyntää rakennuksen digitaalisena kaksosena (Aatsalo, 2018).

Rakennuksen suunnitteluvaiheen jälkeisen valmiin digitaalisen kaksosen pääasiallinen tarkoitus on toimia kiinteistön ylläpidon tietopankkina ja hallinta-alustana, joka sisältää olevan rakennuksen loppudokumenttimuotoiset BIM-mallit, 2D-piirustukset sekä mallin elementteihin syötettyä staattista että dynaamista dataa. Dynaaminen eli elävä data mahdollistaa esimerkiksi huoltotietojen kirjaamisen tilaajalle luovutetun digitaalisen kaksosmallin tietoihin, jolloin kyseinen malli voi toimia myös rakennuksen digitaalisena huoltokirjana. Tällöin rakennuksen huoltohistorian kirjaukset voitaisiin myös tuoda takaisin suunnittelusovellukseen, kun rakennuksen huoltokirjan

digitaalista kaksosta halutaan käyttää inventointimallin pohjana tulevaisuuden korjaussuunnitelmia varten.

2.3 Paljon tietoa kiitos!

Mitä enemmän tietoa, sitä parempi tietomalli. Edellä oleva maininta pitää hyvinkin paikkansa, mutta minkälaisessa tilanteessa se ei pidä? Tällainen tapaus voi olla esimerkiksi tilanne, jossa malliin on kyllä syötetty paljon dataa, mutta kirjattu tieto on toisarvoista tai kokonaan väärää loppukäyttäjän, suunnittelijoiden tiedonvaihdon tai toteutuksen kannalta. Turhaa työtä voi aiheuttaa myös tilaaja/rakennuttaja, joka haluaa tietomallin, mutta ei osaa määritellä sisällön tietotarkkuutta tai -tasoa. Hieman kärjistäen sanottuna malli tilataan edellä mainitussa tapauksessa huvin vuoksi. Tällöin tietomallin käyttö jää törmäyksien, eli suunnittelijoiden mallien yhteen sovittamisen tarkastamiseen ennen rakentamisen aloittamista ja koko muu potentiaali jätetään hyödyntämättä. Tämä on myös osasyy luvussa 2.1.1 mainittuun työmäärän tunteeseen, joka voi aiheuttaa muutosvastarintaa niin suunnittelijoilla kuin työmaatasolla.

Parhaiten muutosvastarintaa voidaan kuitenkin vähentää opastamalla, auttamalla ja kouluttamalla työntekijöitä, joilla on vähemmän kokemusta ja osaamista tietomallityöskentelyn osalta. Tehokkainta opastusta ja auttamista antavat työkaverit, joilla on osaamista kyseisen aihepiirin osalta. Tällöin kynnys kysymiselle pienenee ja tieto pölyttyy tietomalliprojektia työstävien ihmisten kesken. Joillekin myös siirtyminen usean sovelluksen käyttämisestä yhden sovelluksen alle, josta kaikki tieto löytyy, voi toimia ahaa-elämyksen laukaisijana.

Lähtökohtaisesti suurikaan tietomäärä ei ole huono asia. Tärkeintä on, että siitä on hyötyä myös pitkällä aikavälillä, eikä pelkästään projektin alkuvaiheessa. Joskus projekteista voi kuulla puhuttavan ei-tietomalli- ja tietomalliprojektina. Tämä tarkoittaa suunnittelupuolella karkeasti suuntaa mallin sisällä olevalle tietomäärän laajuudelle projektia aloitettaessa. Ajattelutapa on toisaalta järkevä, ettei projektin kannalta tarpeetonta dataa syötetä malliin turhaan. Toisaalta taas se on omiaan lisäämään ylimääräisen työmäärän tunnetta, kun eteen tuleekin

tehtäväksi tietomallipohjainen projekti. Jos jokainen projekti toteutettaisiin pääsääntöisesti alusta loppuun asti tietomallina, niin se vähentäisi tai jopa parhaimmassa tapauksessa voisi hävittää tunteen ylimääräisestä työvaiheesta ja muuttaisi tietomallintamisen rutiininomaiseksi osaksi työskentelyä. Samalla myös harjaantuisi silmä sille, mikä on suunnitelmiin syötettävää tarpeellista informaatiota ja mikä ei.

Viimeistään projekteissa, jotka ovat suuria ja pitkäkestoisia, rutiini tietomallityöskentelyn osalta helpottaa suuren mallin hallintaa ja käsittelyä. Rutiinin avulla voidaan aloittaa tietomallin rakentaminen ensimmäisistä luonnoksista lähtien siihen suuntaan, että tarvittavaa tietoa löytyy riittävästi suunnittelutyön jokaisessa vaiheessa. Alussa tämä lisää toki työmäärää tiedon syöttämisen takia, mutta jos tietoa syötetään tasaisesti projektin edetessä, niin aikaa vapautuu varsinaisten suunnitteluratkaisujen tekemiseen sitä enemmän mitä pidemmälle suunnittelussa edetään. Koska aikaa on aina rajallisesti, niin pahimman paineen aiheuttaa juuri tietomallin tieto-osa, jos sen tekeminen jätetään projektin loppumetreille (Jäväjä & Lehtoviita 2016, s. 78 – 81).

2.4 Tietoa pankista ja pankkiin

Tietomalliprojektin alkaessa tulisi heti tarkastella alustavasti projektiin tarvittava tietosisältö. Pyörää ei välttämättä kannata keksiä uudestaan, vaan dataa voidaan tarpeen mukaan ottaa jo menneistä projekteista. Aikaa tarvitaan alussa myös luonnosteluun ja vaihtoehtojen löytämiseen, joten jos tieto on jo saatavilla valmiina vanhassa projektissa, pienimmän työmäärän tietomallin aloittamisen osalta tuottaa datan tuominen menneestä hankkeesta. Tällöin vaarana voi kuitenkin olla, että tuonnin yhteydessä tulee uuteen projektiin tarpeetonta materiaalia, joka voi mahdollisesti sotkea sisältöä. Tämän välttämiseksi voidaan luoda tarpeellisen tiedon säilyttämiseen niin sanottuja datapankkeja.

Datapankki voi olla esimerkiksi arkisto, joka ei sisällä varsinaisesti mitään mennyttä projektia kokonaisuudessaan, vaan osia useista toteutuneista töistä. Osat voivat olla esimerkiksi suunnitteluratkaisuja, tuotetietoja, tietomallidataa, ominaisuuksia tai mitä tahansa vastaavaa, jonka säilyttäminen nähdään järkeväksi tulevaisuuden tehtävien näkökulmasta. Ajan kuluessa sekä tietomäärän kasvaessa tiedoston sisällä, se voi hyödyttää valtavasti uusien töiden etenemistä. Toistoa vaativat tehtävävaiheet ovat säilöttyinä uusiokäyttöä varten valmiiksi oikeanlaisina. Jotta arkisto olisi kuitenkin luotettava lähde, se tarvitsee vastuuhenkilön tai henkilöitä, jotka hallitsevat sekä ylläpitävät arkistoa. Jos arkiston ylläpito laiminlyödään jatkuvasti, sen päivittäminen kerralla voi olla valtaisa urakka. Ylläpidon laiminlyönti tarkoittaa myös isoja tarkistuskierroksia projekteissa, joihin tietoa ja sisältöä on tuotu vajavaisesta datapankista.

Datapankin ei tarvitse kuitenkaan olla pelkästään yksi arkistotiedosto, vaan se voi olla erilaisten tiedostotyyppien kokonaisuus. Tällöin pankki voi sisältää esimerkiksi aloituspohjan käytetylle suunnitteluohjelmalle, käyttökelpoisia objektikirjastoja, ohjeita sekä kooditiedostoja. Kooditiedostot tosin ovat tällä hetkellä vielä hiljaisena tekijänä taustalla, mutta peilattaessa niitä tulevaisuuteen, niissä on suuri potentiaali toimia itsenäisinä sekä kevyinä datapankkeina, joiden muokattavuus on huomattavasti nopeampaa kuin ison ja raskaan tiedoston. Muokkaus kuitenkin edellyttää pientä paneutumista siihen, kuinka koodi ja sen rakentamiseen käytetty ohjelmointikieli toimii.

3 SUKELLUS OHJELMOINTIIN

Ohjelmoinnin historia on pidempi kuin moni arvaakaan. Ohjelmoitavia laitteita on ollut olemassa jo niinkin aikaisin kuin 800-luvulla. Tuolloin Persialaiset Banu Musan veljekset kehittivät ensimmäisen ohjelmoitavan musiikkisekvensserin, jonka toimintaperiaatteen he kuvailivat kirjassaan ”*Book of Ingenious Devices*”. Sekvensseri ei toki sisältänyt ohjelmaa tai koodia samaan tapaan kuin tämän päivän elektroniset laitteet, tietokoneet tai ohjelmistot vaan sen ohjelmointi perustui enemmän mekaniikkaan ja höyryvoiman luomaan liikkeeseen. (Long ym., 2017)

Ensimmäinen varsinainen koneeseen syötettävä ja kirjoitettu ohjelma syntyi englantilaisen matemaatikon Ada Lovelacen tekemästä työstä. Adan vuonna 1843 kirjoittaman algoritmikoodin avulla voitiin laskea Bernoullin lukujen sekvenssejä Charles Babbagen kehittämässä analyyttisessa koneessa (*analytical engine)*. Tästä alkoi kirjoitetun ohjelmoinnin hiljainen kehittyminen kohti tämän päivän monimuotoisia ohjelmointikieliä. (Tan, 2020)

Vuotta 1954 voidaan pitää seuraavana vedenjakajana, koska tuolloin aloitettiin ensimmäisen varsinaisen korkean tason ohjelmointikielen Fortranin kehittäminen John Backuksen toimesta. Ennen Fortrania ohjelmat kirjoitettiin lähinnä matalan tason ohjelmointikielillä, kuten kone- ja Assembly-kielellä. Fortran ilmestyi jakeluun vuonna 1957.(IBM, 2021)

Apollo-avaruusohjelmalla on ollut osansa ohjelmointikielien kehityksessä. Yhdysvaltalaisen Margaret Hamiltonin työ Apollojen Guidance computer -yksikön ohjelmoinnissa sekä hänen kyseistä yksikköänsä varten kehittämät toimintakonseptit olivat suunnannäyttäjiä ja edelläkävijöitä huippuluotettavien ohjelmistojen suunnittelulle. Margaretin sanotaan käyttäneen ensimmäisenä työnkuvastaan termiä ohjelmistoinsinööri (software engineer). Ennen hänen antamaansa panosta ohjelmistojen kehityksen parissa, niiden suunnitteluun ja kirjoittamiseen ei suhtauduttu yhtä vakavasti kuin ohjelmia käyttävien laitteistojen suunnittelutyöhön. (Hancock, 2014)

Apollo-ohjelman aikaan ja sen jälkeen erilaiset kielet ovat kehittyneet hurjaa vauhtia, ja niitä on tähän päivään mennessä ilmestynyt useita erilaisia. Esimerkkeinä näistä kielistä ovat muun muassa C-kieli, C++, C# (C sharp), Java, Pascal, Basic, VisualBasic ja Python. Jokaisella edellä mainituista kielistä on ollut oma, jonkin erityisen tilanteen luoma käyttötarve ja mahdollisesti alusta, minkä takia kieli on alkujaan kehitetty. Basicin monet voivat tunnistaa esimerkiksi 80-luvulta Commodore- ja Atari-kotitietokoneiden käyttöjärjestelmien ohjelmointi- ja komentokielenä. Python-ohjelmointikielen voi taas tunnistaa esimerkiksi Nokian S60 ohjelmistohankkeesta, jossa *”Python for S60”* nimellä tunnettu paketti oli Nokian oma käännös Python ohjelmointikielestä käytettäväksi S60-alustan ohjelmointiympäristössä (Nokia, 2006).

3.1 Ohjelmoinnin tarpeellisuus arkkitehtisuunnittelussa

Arkkitehdin ammatissa ohjelmointiosaaminen ei tule ensimmäisenä mieleen, kun mietitään ammatin harjoittamisen kannalta välttämättömiä osattavia kokonaisuuksia. Kuitenkin jo tänä päivänä ohjelmointitaitoa vaaditaan esimerkiksi algoritmisessa suunnittelussa, vaikka sitä ei heti välttämättä mieltäisi ohjelmointitaitoa vaativaksi aiheeksi. Algoritmisessa suunnittelussa fokus painottuu geometrian sijaan geometrian muodostavaan logiikkaan. Suunnittelija määrittelee algoritmeihin ehdot ja raja-arvot, joita noudattaen tietokone laskee mahdolliset vaihtoehdot. Algoritmien sekä parametrien avulla voidaan siis luoda erilaisia suunnitteluratkaisuja nopeasti pienellä vaivalla ja tähän suunnittelutapaan liittyvät sovellukset, kuten Rhinoceroksen yhteydessä käytettävä Grasshopper, voivat olla käyttöliittymältään hyvinkin visuaalisia ja käyttäjäystävällisiä. Visuaalisuus on siis vain yksi tapa esittää koodia, eli ohjelmoinnin ei tarvitse olla pelkästään perinteistä tekstipohjaista käskyjen ja muuttujien kirjoittamista, vaan se voi olla hyvinkin intuitiivista. Näistä visuaalisista ohjelmointikielistä puhutaan kirjainyhdistelmällä VPL, joka on lyhenne sanoista *”Visual Programming Language”*.

Mietittäessä kysymystä tarvitsenko arkkitehtinä ohjelmointia mihinkään, vastaus on monitahoinen. Streichin näkemyksen mukaan arkkitehdin tärkein työtehtävä on suunnitella ja hahmottaa ympäristöjä sekä rakennuksia ihmisille, joissa heidän on hyvä olla. Näiden suunnitteluongelmien ratkaisemiseen arkkitehdin täytyy käyttää hänen mieltään, ihmisten välistä kommunikaatiota sekä omaa herkkyyttään. (Streich, 1992) Tähän perustuen voitaisiin tehdä hätäinen johtopäätelmä, että arkkitehti ei tarvitse ohjelmointiosaamista työtehtävissään.

Pysähdyttäessä tarkastelemaan asiaa kuitenkin tarkemmin, vastaus ei olekaan niin mustavalkoinen. Automaation mahdollistava ohjelmointiosaaminen voisi olla arkkitehdille merkittävä taito, jota hän voisi käyttää hyödykseen suunnitteluongelmien ratkaisujen löytämiseen sekä tietojen syöttämiseen digitaaliseen malliin. Ohjelmoinnin kautta säästyvä aika voitaisiin kohdentaa varsinaisiin suunnittelukysymyksiin sekä -ongelmiin sen sijaan, että rajallisia aikaresursseja käytettäisiin suhteessa paljon manuaalista toistoa vaativiin työvaiheisiin.

Tällöin voidaan vastaukseksi sanoa kyllä, arkkitehdin on hyvä osata ohjelmoida sekä ymmärtää sen merkitys työtehtäviensä näkökulmasta. Mitä enemmän ohjelmistot, ohjelmat ja tekniikat kehittyvät, tai suunnitellun muotokielen toteutuksen haastavuus kasvaa, sitä enemmän tarvitaan ohjelmointiosaamista, jotta käytettävissä oleva aika voidaan hyödyntää parhaalla mahdollisella tavalla. Toinen myönteistä vastausta tukeva näkökulma Streichin mukaan on, että tällöin arkkitehti ei ole enää täysin riippuvainen ohjelmistokehittäjien tai tuotevalmistajien tarjoamasta valmiista materiaalista (Streich, 1992).

Streich myös huomauttaa artikkelissaan, että suunnittelutyön rajoitteena eivät saisi toimia käytetystä suunnitteluohjelmistosta johtuvat rajoitteet. Ohjelmoinnin käytettävyyden osalta tulisi näin ollen täyttyä kaksi ehtoa. Ensimmäisenä on vaatimus tarpeeksi hyvin soveltuvasta ohjelmointikielestä käytetyssä CAD-ohjelmassa, joka mahdollistaa yksinkertaisen ohjelmoinnin suunnittelutehtävän ympärillä. Toisena vaatimuksena on se, että suunnittelijan tulisi itse pystyä kirjoittamaan suunnittelutehtävän vaatima koodi toteuttaakseen tarvittavat asiat halutunlaisina tarpeen mukaan. (Streich, 1992).

3.2 Ohjelmoinnin opetus osana arkkitehtiopintoja

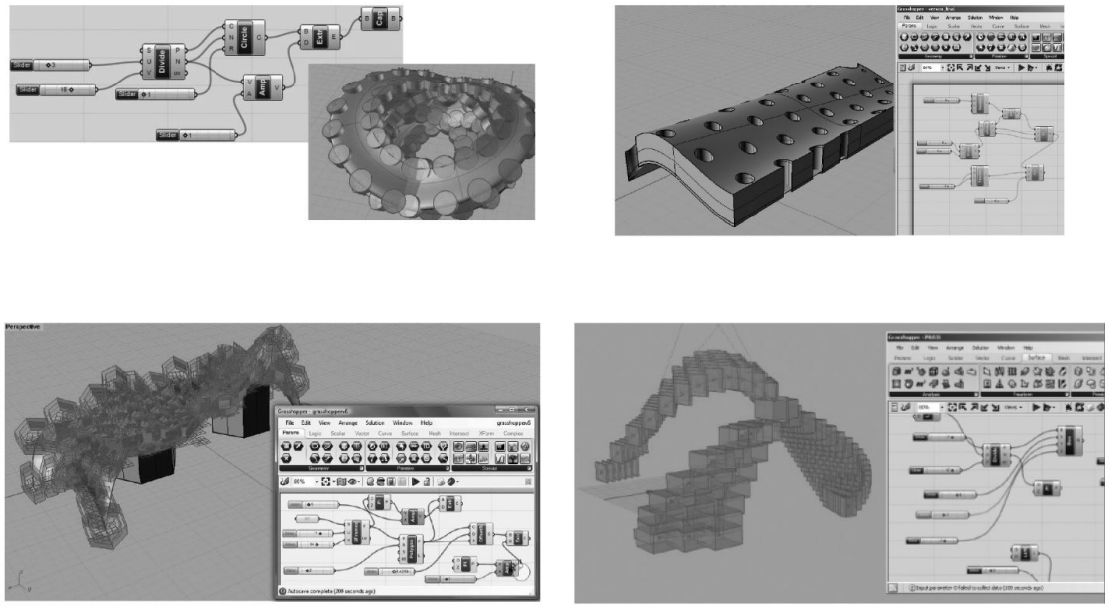
Ohjelmointiopetus on nykyään kiinteä osa peruskoulujen opetussuunnitelmaa. Näin ollen voidaan olettaa, että tulevaisuudessa yhä useamman arkkitehtuuria opiskelemaan pyrkivän henkilön pitäisi osata ohjelmointia jollakin tasolla. Tarkasteltaessa kuitenkin tämän päivän osaamis- ja lähtötasoa voidaan miettiä sitä, minkälainen olisi paras mahdollinen tapa tarjota tieto ohjelmoinnista siitä kiinnostuneelle, mutta ei aikaisempaa kokemusta omaavalle opiskelijalle tai jo valmistuneelle arkkitehdille mahdollisimman ymmärrettävässä muodossa. Vaihtoehtoina ohjelmoinnin opiskeluun ja harjoitteluun ovat käytännössä aikaisemmin mainitut visuaaliset ohjelmointiympäristöt tai perinteisempi tekstipohjainen IDE. Esimerkkeinä opetuksen sisällöstä ja oppimistuloksista nostan esiin Euroopan kolmen eri yliopiston arkkitehtikoulun pitämistä suunnittelukursseista, joiden sisältö painottui ohjelmointiin ja sen hyödyntämiseen suunnittelutyössä.

Tutkimuksia ohjelmoinnin opettamisesta arkkitehtiopiskelijoille on tehty muutamia ja näistä tutkimuksista on saatavilla julkaistuja artikkeleita. Tutkimustyötä aiheesta on tehty esimerkiksi Campinasin, Lissabonin IST ja Istanbulin Mimar Sinan Fine Arts ja Fatih Sultan Mehmet yliopistoissa. Näiden tutkimusten aineistot koottiin kursseille *”CAD in the creative process”* (Campinas), *”Programming for Architecture”* (Lissabon) ja *”Introduction to Parametric Design”* (Istanbul) osallistuneiden opiskelijoiden harjoitustöistä sekä heille osoitetuista kurssikyselyistä.

Celani ja Vaz toteavat artikkelissa *”CAD Scripting and Visual Programming Languages for Implementing Computational Design Concepts: A Comparison from a Pedagogical Point of View.”*, että keskustelu ohjelmoinnin opettamisesta ja sen käyttöönotosta arkkitehtisuunnitteluun yhtenä työkaluna muiden joukossa on ollut konferenssien puheenaiheena varsinkin 1980- ja 1990-lukujen aikana, jolloin tietotekniikka yleistyi voimakkaasti myös arkkitehtien työkaluna. Ohjelmoinnin mahdollisuuksien tutkiminen hiljentyi IT-buumin laannuttua, mutta keskustelu on herännyt uudelleen suunnittelutyöhön saataville tulleiden erilaisten digitaalisten laitteiden ja ohjelmien myötä, jotka mahdollistavat

vapaamuotoisten muotojen sekä massana tehtävän kustomoinnin tarkastelun tavalla, joka ei ole ollut ennen mahdollista. (Celani & Vaz, 2012)

”CAD in the creative process” -kurssi sisälsi historiaa CAD-ohjelmista, toimintaperiaatteet sääntöihin ja parametreihin perustuvista muotoilustrategioista sekä kaksi harjoitustehtävää. Harjoitustehtäviin käytettiin tekstipohjaisena ohjelmointikielenä VBAta (*Visual Basic for Applications*) ja visuaalisena ohjelmointikielenä Grasshopperin tarjoamaa VPLää (*Visual Programming Language*), joka perustuu valmiiden työtilaan valittavien komponenttien kiinnittämiseen toisiinsa solmukohtien avulla. Kummassakin tehtävässä ohjelmointiin liittyvä opetus pidettiin ymmärrettävänä ja yksinkertaisena sisältäen vain yksinkertaisimmat kontrollit, proseduurit ja matemaattiset operaatiot. Oppilaat oppivat nopeasti VPLn, ja pystyivät sen avulla tuottamaan orgaanisia, toistuvia sekä parametrien avulla muuteltavia suunnitelmia (ks. kuva 1).



Kuva 1. Kurssille osallistuneiden opiskelijoiden VPL-harjoitustehtäviä (Celani & Vaz, 2012).

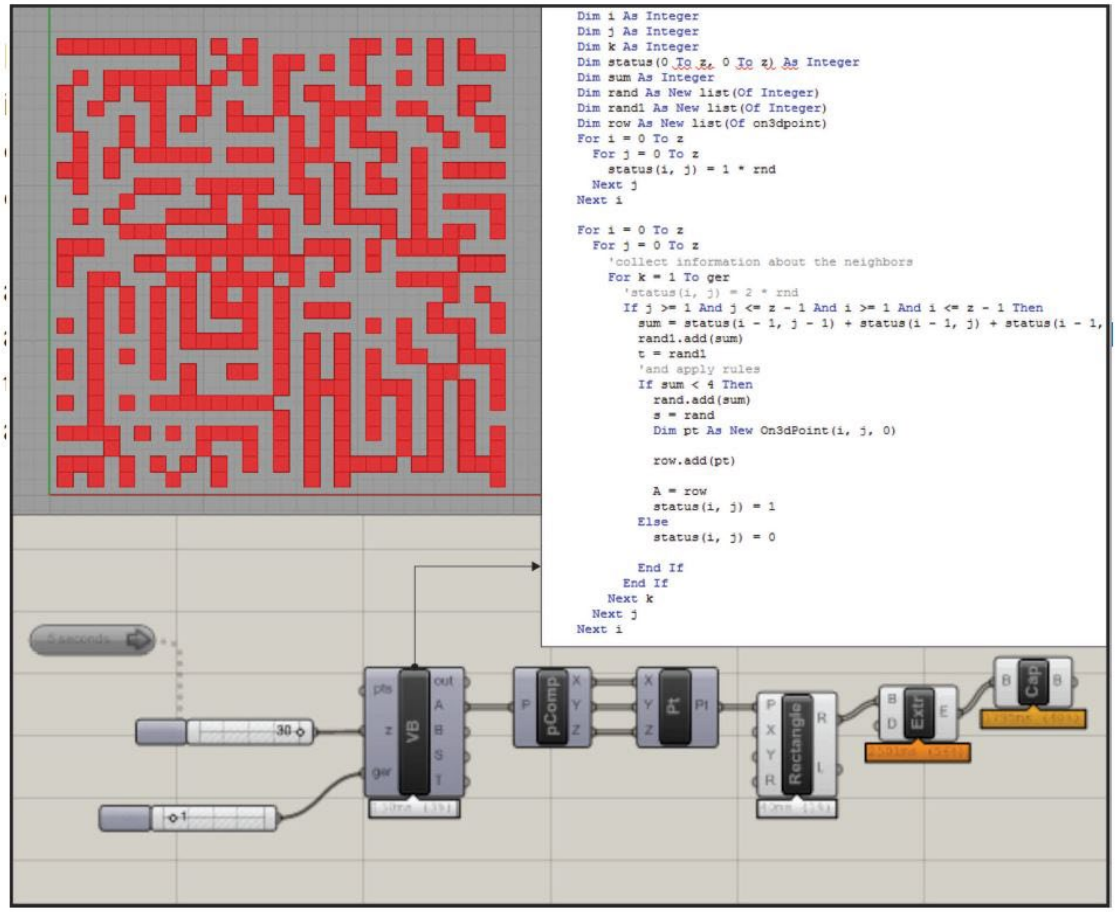
Vertailtaessa eri vuosilta saatuja kurssin oppilaspalautteita, havaittiin että oppilaissa oli tapahtunut asennemuutosta

ohjelmointiosaamisen tarpeellisuuden osalta. Viimeisimmän kurssin oppilaat näkivät visuaalisen ohjelmoinnin hyödyttävän suunnittelutyötä sekä heillä oli halua kokeilla sitä tulevaisuudessa todellisissa työtehtävissä. Aikaisempien vuosikurssien palautteissa oli ilmaistu kiinnostusta aihetta kohtaan, mutta ohjelmointi nähtiin työkaluna, jota he tuskin tulisivat käyttämään työelämässä. (Celani & Vaz, 2012)

Pidemmälle edenneet opiskelijat, jotka valitsivat peruskurssin lisäksi VBA CAD ohjelmointiin painottuvan kurssin, pystyivät tuottamaan huomattavasti monimutkaisempia suunnittelukonsepteja kuin opiskelijat, jotka painottivat ohjelmoinnin opiskelun pelkästään Grasshopperin tarjoamaan visuaaliseen IDEen.

Kirjoitettaessa perinteisemmällä tavalla ohjelmakoodia, oppilaat pystyivät luomaan huomattavasti keveämpiä, tiiviimpiä ja joustavampia muuttujia kuin käyttämällä pelkästään Grasshopperin valmiiksi tarjoamia komponentteja. Varsinkin suunnitelmien koon ja monimutkaisuuden kasvaessa pelkät Grasshopperin tarjoamat komponentit todettiin ongelmallisiksi, koska niitä tarvittiin valtava määrä tavoitellun muodon saavuttamiseksi. Haluttu lopputulos voitiin saada järkevästi vain kirjoittamalla itse tarvittava muuttujakomponentti, jolla voitiin korvata useita valmiita komponentteja (ks. kuva 2, sivu 24). Näin oli mahdollista siistiä sekä yksinkertaistaa Grasshopperissa tehtyä koodin komponenttikaaviota paremmin ymmärrettäväksi kokonaisuudeksi.

Huomiona he kuitenkin painottavat, että suurin heikkous kirjoitetussa koodissa on sen rakenne. Kirjoitetun koodin tulee noudattaa tiukasti ennalta määrättyä tehtäväjärjestystä, kielen merkistöjä sekä syntakseja. Pienimmätkin virheet kirjoitusasussa estävät koodin toimimisen halutulla tavalla. (Celani & Vaz, 2012)

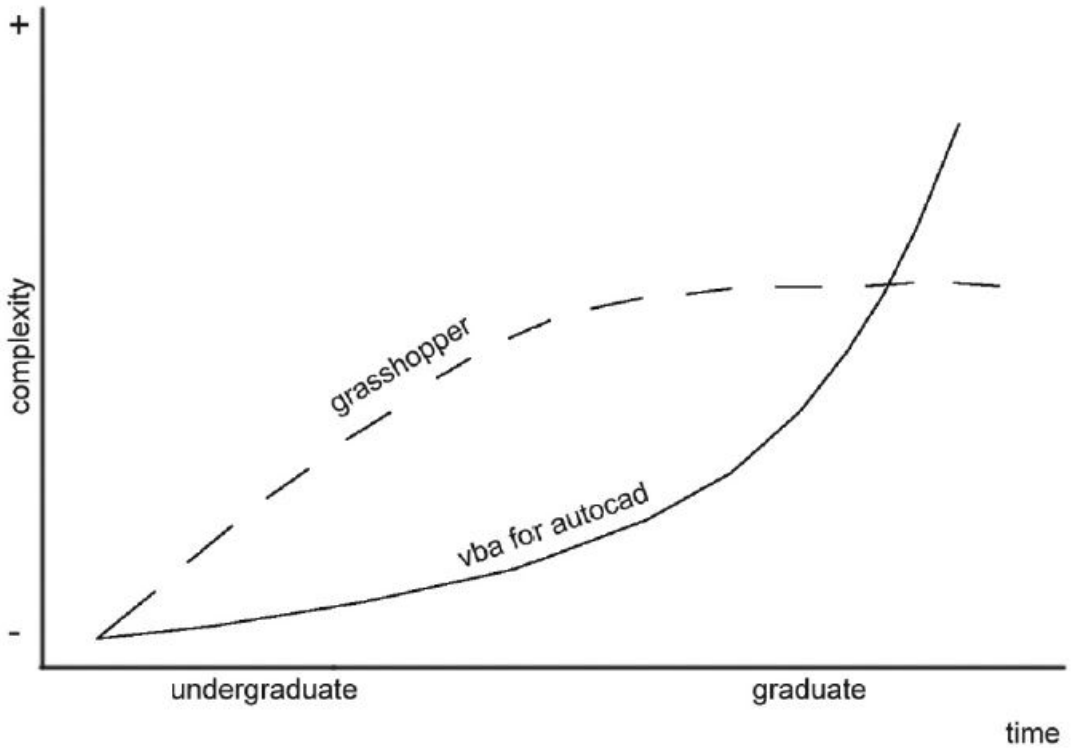


Kuva 2. Carlos Vazin kirjoittama VB-komponentti Grasshopperissa. VB-komponentteja käytettiin tapauksissa, joissa visuaalisen ohjelmointikielen valmiiden peruskomponenttien säädettävyys ei riittänyt halutun muodon toteuttamiseen tarpeeksi yksinkertaisesti ja selkeästi (Celani&Vaz, 2012).

Celani ja Vaz toteavat omien tulostensa perusteella, että vertailtaessa aloittelijoiden tuottamaa tekstipohjaista ja visuaalista koodiaineistoa, jälkimmäinen tuottaa parempia tuloksia oppimisen nopeuden, kielen sisäistämisen ja tuotetun aineiston osalta. Tilanne kuitenkin muuttuu heti kun suunnitelmat monimutkaistuvat (ks. kuva 3, sivu 25). Pelkästään visuaalisen IDEn antamien valmiiden komponenttien ja komentojen käyttäminen rajaa pois monimutkaisempien suunnitelmien ja muotojen suunnittelun sekä tarkastelun.

Tutkimuksen lopputuloksissa nousi esille kaksi mielenkiintoista huomiota. Toinen oli se, että on tärkeää esitellä opiskelijoille ohjelmointitekniikoita, jotta he voivat ilmaista suunnitteluideoitaan

ohjelmoinnin avulla. Toinen tärkeämmäksi koettu huomio oli, että visuaalinen sekä tekstipohjainen ohjelmointiosaaminen auttaa kehittämään opiskelijan kykyä abstraktien käsitteiden ymmärtämisessä, joiden hallinta on erittäin tärkeää arkkitehdin koulutuksessa. (Celani & Vaz, 2012)



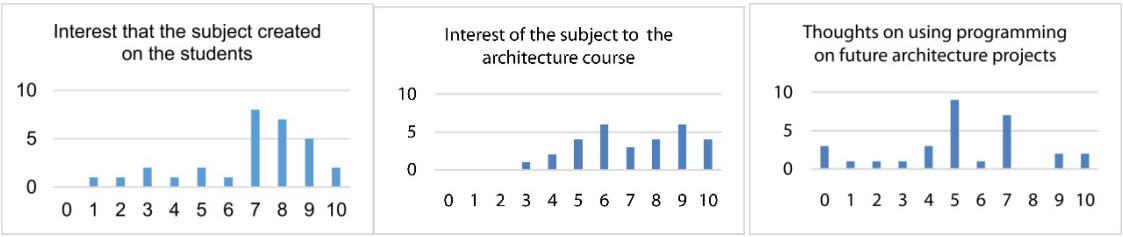
Kuva 3. VBA- ja VPL-ohjelmointikielien oppimiskäyrät kurssilta saatujen tulosten perusteella. Kuvaajasta voidaan nähdä, että ajallisella panostuksella kirjoitettuun ohjelmointikieleen voidaan saada pitkällä aikavälillä tarkempia sekä monimuotoisempia malleja (Celani & Vaz, 2012).

Kurssin *"Programming for Architecture: The Students' Point of View"* ohjelmointiopetuksen pääpaino oli taas perinteisesti kirjoitettavan Racket-ohjelmointikielen opetuksessa, joka on moniparadigmainen ja kuuluu LISP-kielien perheeseen. Kurssin teoriaopetus sisälsi opintoja tietorakenteista, rakentavasta geometriasta, korkeamman asteen funktioista sekä parametrien esitystavoista. Itse ohjelmointia opetettiin vaihtelevien tehtävien avulla, jotka sisälsivät mm. yleisiä

ohjelmointiharjoituksia ja koodausta. Varsinaisten ohjelmien kirjoittamiseen oppilaat käyttivät Racketin natiivia IDEä DrRackettia. Lisänä oppilailla oli käytössään ohjelmakirjasto Rosetta, joka sisälsi ennalta määriteltyjä funktioita sekä toimintoja erinäisiin CAD-sovelluksiin. Rosetta siis varsinaisesti mahdollisti Racket-kielellä tehtyjen ohjelmien ajamisen Autocad- sekä Rhinoceros-sovelluksissa. (Aguiar & Gonçalves, 2015).

Rita Aguiarin ja Afonso Gonçalvesin saamat tulokset olivat vastaavanlaisia kuin Celanin ja Vazin. Niistä kävi ilmi, että suurin osa opiskelijoista oli kiinnostunut ohjelmoinnista aiheena ja sen tuomista mahdollisuuksista, sekä tunnustivat aihepiirin tärkeyden tulevaisuuden näkökulmasta (ks. Kuva 4).

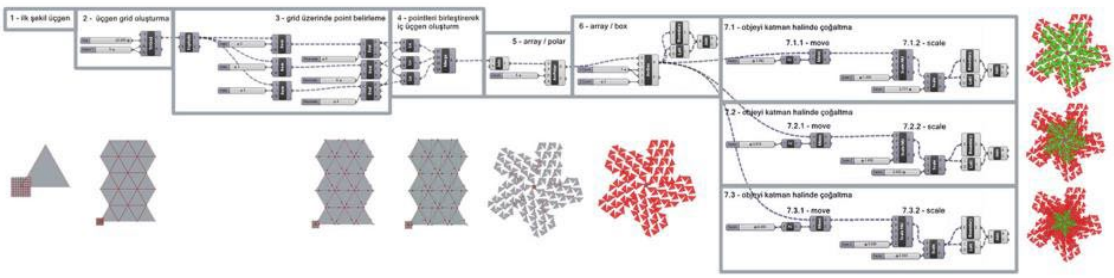
Mutta kuten Celanin ja Vazin artikkelissa, opiskelijat arvioivat, etteivät tulisi käyttämään ohjelmointia jatkossa. Syyksi oppilaat ilmoittivat sen, että muilla suunnittelukursseilla ei rohkaistu tarpeeksi uusien toimintatapojen kokeilemiseen tai niiden itsenäiseen tutkimiseen. Lopputyössä haasteen loisivat taas puutteelliset ajalliset resurssit (Aguiar & Gonçalves, 2015). Tästä voidaan karkeasti päätellä, että jotta ohjelmointi tulisi hyödylliseksi osaksi arkkitehdin työkalupakkia, sitä tulisi pyrkiä sisällyttämään jossain muodossa jo opiskeluvaiheessa mahdollisimman laaja-alaisesti eri kursseilla.



Kuva 4. Oppilaskyselyjen tuloksia. Kuvaajien grafiikoissa on esitetty pylväsdiagrammein ohjelmoinnin kiinnostavuus aiheena sekä opiskelijoiden arviot ohjelmoinnin käytöstä heidän tulevaisuuden projekteissaan. Käytetty arviointiasteikko oli tutkimuksen kysymyksissä 0–10 (Aguiar & Gonçalves, 2015).

”Teaching Design by Coding in Architecture Undergraduate Education” – kurssilla ei opetettu varsinaista kirjoitettavaa ohjelmointikieltä vaan opiskelijoille pyrittiin avaamaan ohjelmoinnin logiikkaa. Heille annettiin tehtäväksi suunnitella jokin Islamilainen kuvio. Aluksi opiskelijoiden tuli tutustua kuvion rakenteeseen ja muodostaa näkemys siitä, kuinka kuvion voisi rakentaa matemaattisesti ja tämän perusteella toteuttaa se matemaattisiin sääntöihin perustuen. Kuvioiden rakentaminen aloitettiin hahmottamalla lähtösääntö peruskuviolle Grasshopperissa, jonka pohjalta edettiin sääntökokonaisuus kerrallaan kohti monimutkaisempaa kuvion toistorakennetta. (Agirbas, 2017)

”Teaching Design by Coding in Architecture Undergraduate Education” – kurssin tuloksista paljastui oppimisen kannalta merkittävä huomio. Kävi ilmi, että kuvion ohjelman rakenteen pilkkominen sääntökokonaisuuksiksi helpotti ohjelmoinnin oppimista ja sen sisäistämistä. Oppilaille kehittyi kurssin aikana ymmärrys siitä missä järjestyksessä heidän tulee lähteä purkamaan annettua ongelmaa ja rakentamaan sille ratkaisua. Kurssin edetessä he pystyivät nopeasti tarkentamaan ja laajentamaan suunnitelmiaan sekä luomaan tarpeellisia sääntöjä, jotka oppilaat nimesivät välipisteiksi (”*breakpoint*”). Nämä välipisteet muodostivat suunnitelmien rungon. Niiden avulla oppilaat pystyivät helposti palaamaan tarvittaessa kuvion lähtötilanteeseen tai toteuttamaan kokonaan uuden kuvion muuttamalla sääntökokonaisuuksien sijainteja sekä parametreja (ks. kuva 5). (Agirbas, 2017)



Kuva 5. Kurssille osallistuneen oppilaan tekemä, sääntökokonaisuuksin perustuva ohjelma, joka muodostaa parametrien perusteella halutun kuvion määritellystä lähtömuodosta (Agirbas, 2017).

Vaikka kurssilla opiskelijoille ei opetettu varsinasta tekstipohjaista ohjelmakoodin kirjoittamista, niin opetettu ajattelu- sekä lähestymistapa annetun ongelman ratkaisemiseen pätevät myös kirjoitettavissa ohjelmointikielissä. Sääntökokonaisuudet helpottavat ymmärtämään kielen toimintaa, rakennetta sekä järjestyksen luomista ohjelman vaatimalle koodille.

3.3 Arkkitehtisuunnittelun ohjelmointikielet

Ohjelmointitaidon hyödyntäminen arkkitehtisuunnittelussa on ollut mahdollista jossain muodossa jo 1980-luvulta lähtien. Tuolloin LISP-kieltä käyttäen voitiin kirjoittaa tekstimuotoisia ohjelmia, joiden käskyrivien komentojen mukaan haluttu geometria on piirtynyt sovelluksen mallitilaan. Tänä päivänä näkyvin ohjelmointikielityyppi arkkitehtisuunnittelussa on VPL eli visuaaliset ohjelmointikielet. Tätä ohjelmointitapaa on mahdollista käyttää esimerkiksi Grasshopperissa, Autodeskin Dynamossa, Archicadin PARAM-O työkalussa sekä Unreal Engine -pelimoottorissa. Unreal Engine on arkkitehtuurin piirissä hieman tuntemattomampi sen pääkäyttötarkoituksen painottuessa pelimaailmojen suunnitteluun. Kyseistä sovellusta käytetään tästä huolimatta myös arkkitehtuurin saralla virtuaalimallien rakentamiseen, kun on tarve luoda hyvin interaktiivisia sekä vuorovaikutteisia ympäristöjä sekä malleja.

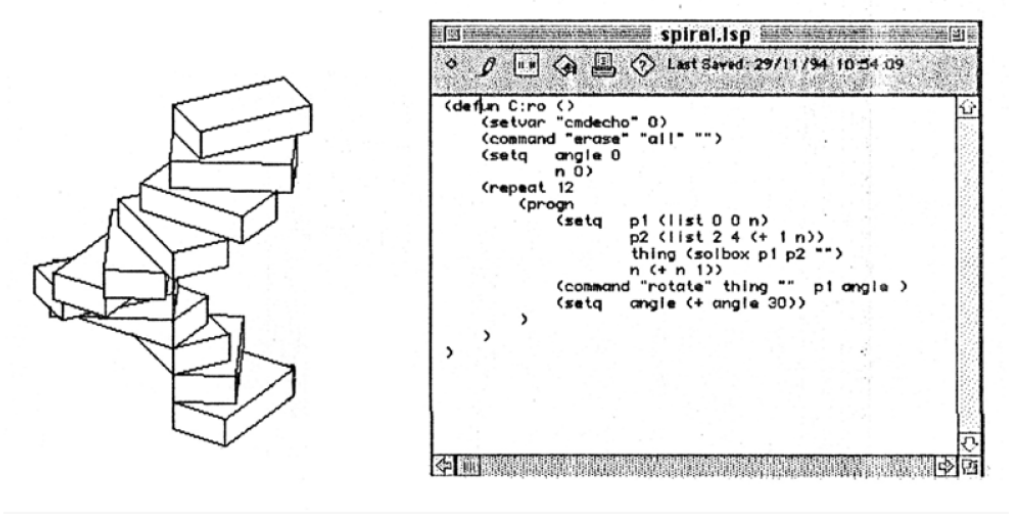
3.3.1 AutoCAD VBA ja AutoLISP

Autodesk-tuoteperheen AutoCAD sovelluksen sisällä on ollut käytettävissä AutoLISP niminen LISP-ohjelmointikielen murre *(”dialect”)* vuodesta 1986 lähtien. AutoLISP on pieni, dynaamisesti kirjoitettu kieli, jossa on automaattinen roskienkeräys, muuttumaton luettelorakenne ja asetettavat symbolit, josta puuttuvat tavalliset LISP-ominaisuudet, kuten makrojärjestelmä, tietueiden määritystoiminnot, taulukot sekä funktiot vaihtelevalla määrällä argumentteja. Lisäksi AutoLISPin useimmat primitiiviset toiminnot (kuten boolean toiminnot ”*and, or,*

not”, Aritmeettiset toiminnot *”+, -, *, <,>”* jne.) ovat geometriaa, AutoCADin sisäisen DWG-tietokannan käyttöä tai graafisten kokonaisuuksien käsittelyä varten.

AutoCAD tukee myös VBA- (*Visual Basic for Applications*) ohjelmointikieltä, joka perustuu olio-ohjelmointiin (*”object oriented programming”*). VBA-koodit, joita kutsutaan myös makroiksi, voidaan kirjoittaa käyttämällä VBAIDEä tai erikseen AutoCADiin ladattavaa lisäosaa. AutoLISPiin nähden VBA tarjoaa myös käyttäjäystävällisemmän sekä intuitiivisemman käyttöympäristön. Celani ja Vaz kirjoittavat artikkelissaan, että vaikka kieli ei mahdollista uusien varsinaisten olioluokkien tai itsenäisten ohjelmien tuottamista, se on hyvin tehokas automatiikkaa vaativiin tehtäviin AutoCADin sisällä. Kielellä on tyypilliset ehdot (*”if”, ”then”, ”else”*) ja toistot (*”for”, ”each”, ”next”, ”dowhile”, ”goto”*) kuten muissakin ohjelmointikielissä. Sen proseduurit voidaan ryhmittää funktioiden tai aliohjelmien alle. (Celani & Vaz, 2012)

Kirjassa *”Microcomputer Aided Design: For Architects and Designers”* Gerhard Schmitt (1988) käytti esimerkeissään AutoLISPillä (ks. kuva 6, sivu 30) kirjoitettuja koodeja ja totesi niiden avulla ohjelmoitavan tietokoneavusteisen suunnitteluprosessin edut: Suuren tietomäärän saatavuuden valmiista tietokannoista sekä mahdollisuudet käyttää konepäätelmiä ratkaisemaan uusien suunnitteluun liittyvien ongelmien aiempien suunnitteluratkaisujen pohjalta. Lopuksi suunnittelijan tehtävänä olisi lopulta vain tarkastella kriittisesti koneellisesti luotuja vaihtoehtoja. Edellä mainitusta Schmittin näkemyksestä on Celanin mukaan tulkittavissa, että arkkitehtien tulisi opetella ohjelmoimaan, jotta he voisivat työtehtävissään valjastaa käyttöönsä suunnittelusovellusten koko potentiaalin. (Schmitt, 1988; Celani, 2008)



Kuva 6. Schmittin AutoLISPillä tekemä koodiesimerkki (Schmitt, 1988; Celani, 2008).

3.3.2 Archicad GDL, PARAM-O ja lausekkeet

Archicadin puolella ohjelmointiosaamista on voinut hyödyntää GDL-kielen *(”Geometric description language”)* parissa, jonka rakenne pohjautuu BASIC-ohjelmointikieleen. Kummankin kielen lausekkeiden marssijärjestys sekä muuttujien logiikka on samanlainen. GDL-kielen avulla voidaan luoda parametreihin perustuvia objekteja sekä objektin ″käyttöliittymä″. Samassa yhteydessä tekstipohjainen koodi rakentaa valmiiksi kirjoitettuihin staattisiin ja jälkikäteen muutettaviin dynaamisiin parametreihin perustuen objektin 2D- ja 3D-näkymät. GDL-kieli on kuitenkin jäänyt nichemäiseksi, eli pienen ryhmän mielenkiinnon kohteeksi, koska kieltä ei ole käytössä missään muussa sovelluksessa. Peruskäyttäjän tarve sen käyttämiselle Archicadissa voi olla vain harvakseltaan tai jopa ei ollenkaan. Sen toimintaperiaate on myös haasteellisempi opetella vähäisen kirjallisessa muodossa olevan opetusdokumentaation takia.

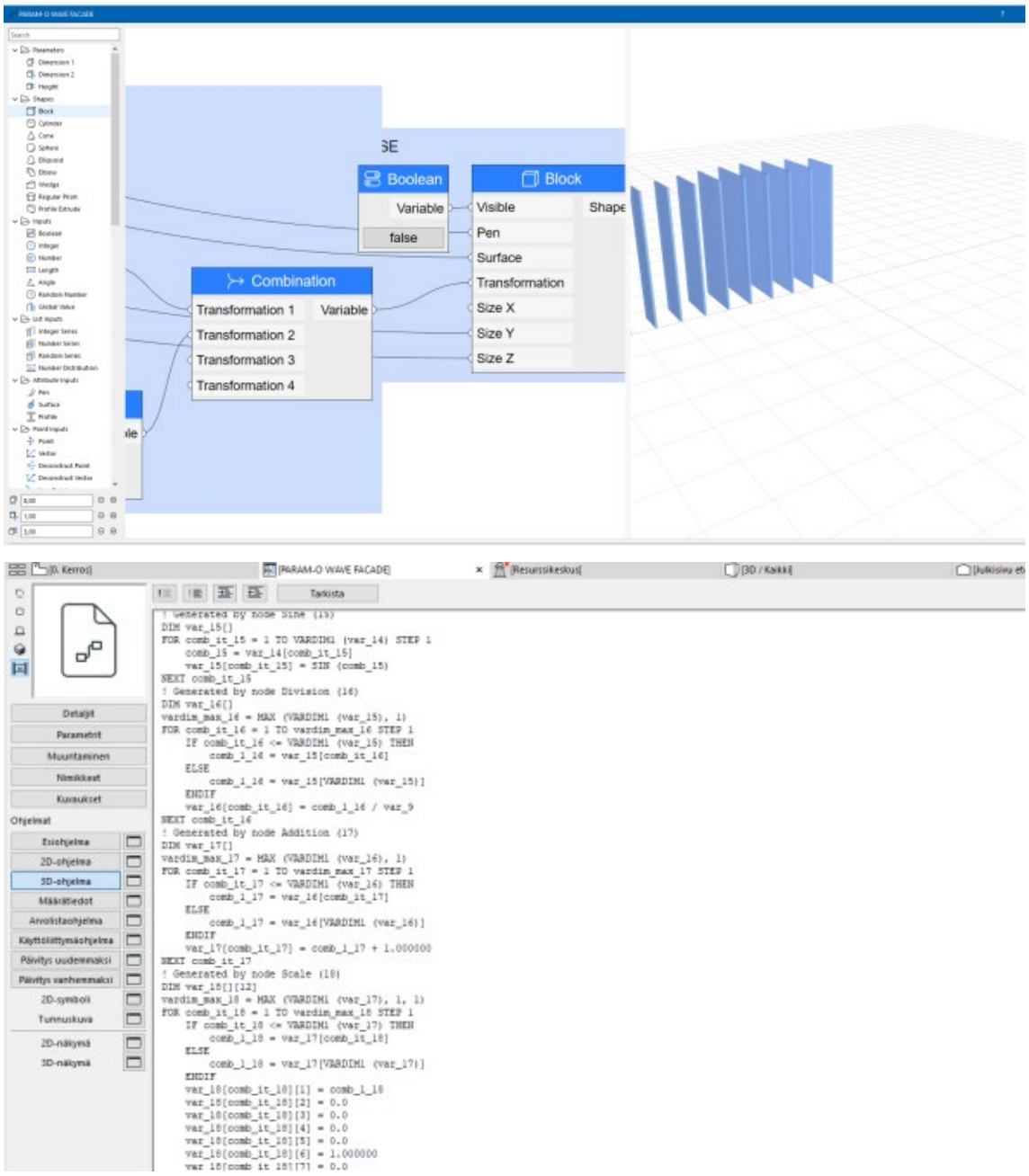
Ongelmalliseksi valmiit GDL-kielipohjaiset objektit voivat muodostua tilanteessa, jossa niiden muokkausta pitäisi pystyä tekemään

paikallisesti. GDL-objekti voidaan lukita salasanoin, jolloin niiden käyttäjäkohtainen muokkaus on käytännössä mahdotonta. Vaikka objektin suunnitelmiinsa tuova suunnittelija osaisikin GDL-kielen, niin mahdollisesti tarpeellinen objektiin koodiin tehtävä korjaus tai elementin parametrien laajentaminen on estetty kirjaston tuottajan liiketoiminnallisen näkökulman takia.

Huolimatta GDL-kielen haastavammasta käyttöympäristöstä, sen merkitys on kuitenkin kriittinen päivittäisen suunnittelutyön kannalta Archicad-työympäristössä. Ilman tuon kielen osaajia, objektikirjastojen määrä olisi huomattavasti nykyistä määrää suppeampi. Toisaalta taas käytetyn kielen takia, kolmansien osapuolien tuottamien objektikirjastojen määrä on Archicadissa huomattavasti pienempi kuin REVIT-ohjelmalle saatavilla olevat kirjastot.

Tarpeen vaatiessa, visuaalinenkin ohjelmointi on Archicadissä mahdollista PARAM-On avulla (ks. Kuva 7, sivu 32). PARAM-O oli Archicadin versioon 24 asti erikseen ladattava lisäosa ja versiosta 25 lähtien se on kiinteä osa kyseistä ohjelmaa. PARAM-On avulla voi tämän diplomityön kirjoitushetkellä toteuttaa yksinkertaisia omia objekteja Archidadissä työstettävän projektin sisällä. Tällaisia omia objekteja voivat olla esimerkiksi pöydät ja tuolit, joiden skaalattavuutta halutaan muuttaa objektin asetuksista ilman, että kyseistä objektia tarvitsisi muuttaa kooditasolla jälkikäteen.

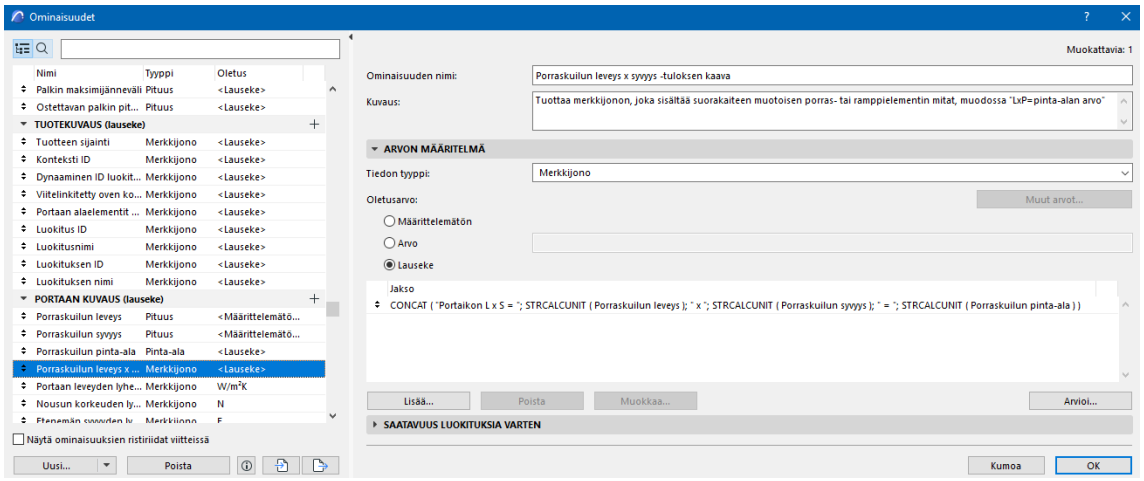
PARAM-O hyödyntää vastaavaa ohjelmointiympäristön rakennetta objektien luomiseen kuin 3. luvun alussa mainittu Grasshopper. Vaikka vielä tällä hetkellä työkalussa on tiettyjä rajoittavia reunaehtoja, se voi tulevaisuudessa kuitenkin syrjäyttää GDL-pohjaisen koodaamisen päivittäisessä käytössä paremman sekä havainnollisemman käyttöliittymän ansiosta. Työkalun toiminnallisuuden laajentaminen mahdollistaisi sen, että arkkitehti tai suunnittelija voisi rakentaa itse paremmin tarvitsemansa objektit hyvinkin ketterästi.



Kuva 7. Sama objekti, kaksi eri tapaa koodata. Alla perinteinen GDL ohjelmointiympäristö. Ylempänä PARAM-On visuaalinen ohjelmointiympäristö.

Helpommin lähestyttävää sekä päivittäiseen tarpeeseen sopivaa ohjelmointitapaa on kehitetty osaksi Archicadin ominaisuuksia versiosta 22 lähtien. Ohjelmointikielellä ei ole varsinaista nimeä, vaan sen voi tunnistaa Archicadin ominaisuuksissa merkinnästä ”*Lauseke*” (Eng. ”*Expression*”) (ks. Kuva 8). Tuossa ohjelmointikielessä on

käytettävissä muutamia funktioita, joiden avulla voidaan muuttaa tai yhdistää elementtien tietoja haluttuun muotoon. Sen hyödyksi voidaan katsoa laaja dokumentaatio ja osaamisen kartuttaminen. Vaikka kyseinen ohjelmointitapa on käytössä vain tietyn sovelluksen sisällä, sen avulla voi ymmärtää ohjelmointikielen toimintaa sekä logiikkaa helposti lähestyttävällä tavalla.



Kuva 8. Lausekkeen periaate Archicadin sisällä. Kuvan lauseke yhdistää tiedon porraskuilun leveydestä, syvyydestä sekä peittopinta-alasta. Kyseisessä lausekkeessa mallista tulevat tiedot muutetaan ensin laskennallisista yksiköistä tavallisiksi merkeiksi, jotta useampaa erilaista mitattavaa yksikkötietoa voidaan yhdistää yhteen solukenttään. Kuvan esimerkkilauseke löytyy suomenkielisestä Archicadin aloituspohjasta.

3.3.3 Arkkitehtisuunnitteluohjelmiin soveltuvat ”yleiskielet”

Kaikkia arkkitehtisuunnittelusovelluksissa tuettuja ohjelmointikieliä ei kuitenkaan rajoita itse suunnitteluun käytetty suunnitteluohjelma. Tällaisia ohjelmointikieliä, joista on saatavilla riittävää dokumentaatiota ja joita voidaan käyttää suhteellisen vapaasti halutussa suunnittelusovelluksessa, kaksi kieltä nousee selkeästi esiin suuresta joukosta. Nuo kaksi ovat C-kielen eri variaatiot ja Python. Kummatkin kielet kirjoitetaan tekstipohjaisesti, halutussa IDEssä. IDE ei ole kuitenkaan välttämätön sillä tekstipohjaisen koodin kirjoittaminen on mahdollista tehdä mitä tahansa tekstieditoria käyttäen. Tällöin

kuitenkin suurin osa tarpeellisista IDEn tarjoamista toiminnoista, kuten esimerkiksi virheentarkistus tai koodin ulkoasun helpompi hallinta ei ole mahdollista.

Arkkitehtisuunnittelun osalta päivittäistä käyttöä tarkastellen C- ja Python-kielen käyttökohteet eroavat kuitenkin toisistaan huomattavasti, mikä johtuu niiden toimintaperiaatteista. C-kieli ja sen eri variaatiot soveltuvat esimerkiksi suunnittelusovelluksien osalta erilaisten laajennusosien kirjoittamiseen. Näiden laajennusosien avulla voidaan käytetyssä suunnittelusovelluksessa muuttaa esimerkiksi elementtien tietoa automaattisesti melko syvältäkin tai tuoda numeerista dataa, jonka laajennusosa muuttaa ennalta määritellyksi elementiksi. Python vastaa taas tarpeeseen, jossa halutaan toteuttaa nopeasti tiedon syöttäminen sekä noutaminen elementeistä niiden tyypistä riippumatta ilman erillistä laajennusosaa tai sijoittaa käytetyn suunnittelusovelluksen valmiita elementtejä määrättyihin pisteisiin mallitilassa.

C-kielen ja varsinkin C-kielen C++ version käyttäminen jouhevasti arkkitehtisuunnittelussa vaatii kuitenkin huomattavasti enemmän paneutumista ohjelmoinnin periaatteisiin sekä mielellään aikaisempaa ohjelmointikokemusta, kun taas Python on rakenteeltaan aloittelijaystävällisempi. C-kieli tarvitsee myös kääntää ennen kuin sitä voidaan ajaa halutussa ohjelmistossa. Python-koodi voidaan ajaa sellaisenaan suoraan käytetyssä, Python-tiedostoja tukevassa ohjelmistossa. Netistä on myös löydettävissä useita valmiiksi Python-kielellä tehtyjä ohjelmia arkkitehtisuunnittelusovelluksiin, joita käyttäjät voivat itse tutkia sekä muokata. Pythonilla on siis yksi valttikortti hihassaan. Se on laaja kehittäjäyhteisö, jota ohjelmointipiireissä kutsutaan sanalla ”*community*”. Pythonin itseopiskelua helpottaa myös kattava sekä helposti saatavissa oleva kirjallinen lähdeaineisto.

3.4 Python lyhyesti

Python-kieli näki päivänvalon 1980-luvulla ja sen ensimmäinen versio julkaistiin joulukuussa 1989. Kielen luoja, hollantilainen Guido Van Rossum kehitti Pythonin ABC-ohjelmointikielen perilliseksi ja jatkoi sen pääkehittäjän toimenkuvassa aina vuoden 2018 heinäkuuhun asti. Nimensä Python sai humoristisesti Brittiläisen Monty Python komediaryhmän mukaan. Nimi koettiin tarpeeksi lyhyeksi ja yksilöiväksi, jotta kieli olisi helposti tunnistettavissa. Komedia ja huumori ovat periytyneet myös kielen dokumentaatioon, koska sen koodiesimerkeissä pyritään välttämään liian vakavaa ja ehkä hieman insinöörimäistä ilmaisua satunnaisilla viittauksilla Monty Pythonin tuotantoon. (Klein, 2021)

Python on kehitetty moniparadigma- ohjelmointikieleksi. Ohjelmointiparadigmalla tarkoitetaan varsinaisen ohjelmointikielen taustalla olevaa perustavanlaatuista tapaa ajatella ja mallintaa ohjelmointitehtävän ratkaisu. Toisin sanoen se on tapa toteuttaa itse varsinainen tietokoneohjelma. Paradigmat eroavat toisistaan riippuen siitä mistä eri osista ohjelma koostuu sekä miten hallinnan ja laskennan eteneminen ilmaistaan. Osia ovat esimerkiksi oliot, funktiot, muuttujat ja niin edelleen. Python-kielessä paradigmoja ovat olio-, proseduraalinen- sekä funktionaalinen-paradigma (Boudreau, 2020). Edellä mainitut paradigmat määrittävät siis sen miksi Python-ohjelmointikieltä kutsutaan moniparadigmakieleksi.

Python käyttää dynaamista tyypitystä sekä viitelaskennan ja roskienkeruun yhdistelmää. Dynaaminen tyypitys tarkoittaa sitä, että tyyppi tarkistetaan ja päätetään ohjelman ajon aikana. Vastaavasti staattisessa tyypityksessä tyyppisidos päätetään vasta käännösvaiheessa. Sen sijaan, että kaikki toiminnot olisivat rakennettuina suoraan kielen ytimeen, Python on suunniteltu siten, että se on modulaarinen sekä helposti laajennettavissa (Python, 2021). Van Rossumin näkemys pienestä ydinkielestä, jossa on suuri standardikirjasto ja helposti laajennettava tulkki, johtui hänen turhautumisestaan ABC:tä kohtaan, joka toimi tältä osin täysin päinvastaisella tavalla kuin Python (Klein, 2021). Modulaarisuus mahdollistaa esimerkiksi ohjelmoitavien liitäntöjen lisäämisen jo olemassa oleviin sovelluksiin. Tämän modulaarisuuden

ansiosta Python on voinut laajentua helposti osaksi arkkitehdeille suunnattuja suunnittelusovelluksia. Esimerkiksi Rhinoceros3D+Grasshopper, Revit- sekä Archicad-ohjelmassa on mahdollista ajaa Python-kielellä kirjoitettuja ohjelmia.

3.4.1 Automatisoitu toisto

Suurissa tietomalliprojekteissa manuaalisen työn määrä on valtaisa. Siksi ne ovat parhaita kohteita kokeilla muuttaa toistoa vaativat työtehtävät automatisoiduiksi kirjoittamalla kyseisen työvaiheen suorittava ohjelma. Tällainen tehtävä voi olla esimerkiksi jonkin tiedon syöttäminen elementteihin (paloluokka, ääniluokka jne.), tyyppielementin sijoittelun/muodon muokkaus tai yhden elementin ympärille toisenlaisesta elementistä rakentuva kokonaisuus (esimerkiksi ikkunan geometrian suhde sen ympärillä olevan julkisivuverhouksen rytmitykseen). Pelkkään tiedon syöttämiseen riittää itse .py-koodi, joka sisältää määritykset ja lopputuleman. Siirryttäessä elementtien muotoa sekä siirtoa vaativiin toimintoihin, tarvitaan vielä tällä hetkellä lisäksi VPL-ohjelma tai erikseen työvaihetta varten kirjoitettu lisäosa.

Ilman aikaisempaa ohjelmointikokemusta koodiin tutustuminen ja ensimmäisen ohjelman kirjoittaminen vie yleensä enemmän aikaa kuin edessä olevan tehtävän tekeminen perinteisempään tapaan manuaalisesti. Ajallisesta näkökulmasta tarkasteltuna osat kuitenkin vaihtuvat, kun yhtä aikaa työstettävien projektien koot ja määrät kasvavat. Manuaalinen tapa kuluttaakin suuren määrän työtunteja ”turhaan” sellaisiin tehtäviin, jotka toistuvat sisällöllisesti samanlaisina projektista toiseen. Oman lisänsä tähän työmäärään tuovat vielä sellaiset suuret projektit, joissa suurin osa tietosisällöstä joudutaan muuttamaan kesken suunnittelutyön. Tällöin muutoksesta aiheutuva lisätyö suunnittelijalle voi olla hyvinkin merkittävä ja aikaa vievä tarkastuskierroksineen. Kerran kirjoitettua ohjelmaa voidaan kuitenkin käyttää taas tehokkaasti uudestaan. Valmiiseen ohjelmakokonaisuuteen voidaan nopeasti ja helposti tehdä muutoksia tai lisätä uusia osia projektin sisällön niin vaatiessa. Ohjelmia voidaan kirjoittaa myös sellaisia tarkastuksia ja tarkasteluja

varten, jotka on aikaisemmin tarkistettu elementti kerrallaan silmämääräisesti.

Python-kielen tehokasta käyttämistä varten tulee kuitenkin tuntea jossain määrin käytetystä suunnittelusovelluksesta johtuvat rajoitteet. Saatavilla olevien suunnitteluohjelmien kesken on paljon vaihtelua siinä, kuinka paljon Python-kielelle on avattu rajapintaa keskustelua ja tiedonvaihtoa varten käytetyn suunnittelusovelluksen työkaluihin. Helpoimmin rajoitteet voidaan saada selville tutkimalla sovelluksen kirjallista *”API Python”* -dokumentaatiota. Näissä dokumenteissa on listattu tuetut komennot sekä niiden toiminnallinen kuvaus.

Sovelluksesta johtuvia rajoitteita on mahdollista kiertää kirjoittamalla ohjelmiin erikseen lisättäviä lisäosia, jotka aukaisevat lisää rajapintaa suunnittelusovelluksen ja Python-kielen välille. Lisäosien kirjoittamiseen itsenäisesti vaaditaan kuitenkin yleensä laajempaa ohjelmointiosaamista. Tällaisesta itse kirjoitetusta lisäosasta, joka hyödyntää suunnittelusovelluksen puolella erillistä Python-koodia, hyvänä esimerkkinä voidaan nostaa esille Gui Talaricon 20th Pyninsula Meetingissä esittelemää työpisteiden optimaalisinta sijoittelua tarkastelevaa ohjelmaa (Pyninsula Official, 2019). Lähtötilanteen ajossa määritellään tilaa rajaavat muodot, jonka perusteella ohjelma sijoittaa tilaan mahtuvat työpisteet niin, että niitä saadaan optimaalinen määrä huomioiden tilassa vaaditun sisäisen liikenteen viemä osuus. Ohjelman antama ehdotus oli nähtävissä mallitilassa välittömästi tarvittavien alkumääritysten jälkeen.

Edellä mainittu ohjelma perustuu käytännössä erilaisiin laskukaavoihin, jotka aktivoituvat, kun tarvittavat lähtötiedot on syötetty koodin alkuparametreiksi. Vaikka tehtävä itsessään on yksinkertainen ja selkeä, vastaavanlaisen tarkastelun suorittaminen ihmissilmin on huomattavasti haastavampaa ja aikaa vievämpää. Tällöin joudutaan tarkastelua varten tekemään manuaalisesti yksi vaihtoehto kerrallaan ja pitämään aikaisemmin luodut vaihtoehdot muistissa vaihtoehtojen keskinäistä vertailua varten.

3.4.2 Ulkoasu ja rakenne

Avatessa ensimmäistä kertaa .py-päätteistä tiedostoa käy selväksi, että Python-ohjelma näyttää periaatteessa visuaalisesti samalta kuin mikä tahansa muu kirjoitettava ohjelmointikieli. Ilman aikaisempaa kokemusta ohjelmoinnista Pythonilla kirjoitetusta ohjelmasta ei käy ilmi sen toimintaperiaate, koska ymmärrys ohjelman rakenteesta puuttuu. Tämän takia tarkastelu kannattaa aloittaa lyhyemmistä ja selkeämmistä koodikokonaisuuksista tai jopa yksinkertaisista harjoitustehtävistä. Mitä enemmän kirjoitettuja ohjelmia lukee ja niitä kirjoittaa itse, voi alkaa näkemään toiminnallisuuteen vaikuttavat osat sekä toiminnan rakenteen.

Yksinkertaisin harjoitustehtävä on *”Hello, world!”* -ohjelma. Se on myös ensimmäinen ohjelma, joka opetetaan ohjelmointia käsittelevillä kursseilla tai itseopiskeluun tarkoitetuilla sivustoilla. Sivustoilla ei kuitenkaan avata sitä kuinka jo pienet muutokset *”Hello, world!”*-ohjelman koodissa vaikuttavat sen toimivuuteen. Ohjelmalla on kuitenkin hyvin helppo havainnollistaa kielen toiminnallisia pääperiaatteita, ja siksi päätin käydä läpi tämän pienen kokonaisuuden useammasta kulmasta. Python-kielellä ohjelma kirjoitetaan seuraavasti: *print(”Hello, World!”)*. Ohjelman *print*-komento tulostaa sulkumerkkien sisällä olevan sisällön komentoriville (ks. Kuva 9).

Kuva 9. Vasemmalla työskentelytila, oikealla komentorivi johon sisältö tulostuu. Komento ”print” tulostaa tekstin ”Hello, World!” merkkijonona komentoriville.

Sen kuinka asiat ilmaistaan sulkumerkkien sisällä, täytyy noudattaa määrättyä rakennetta. *”Hello, world!”* -ohjelma ei sisällä esimerkiksi noudettavia muuttujia tai listatietoja, vaan pelkästään erilaisia

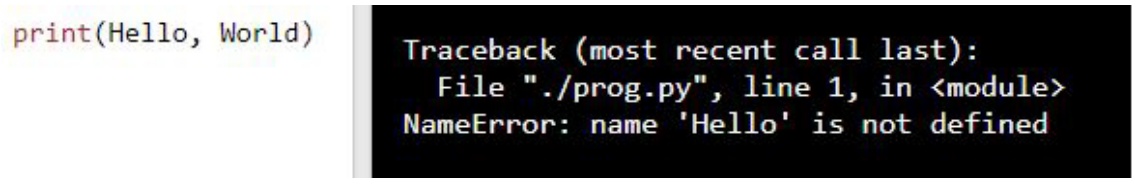
merkkejä eli merkkijonon. Tällöin alkuun ja loppuun tarvitaan lainausmerkit, joilla osoitetaan tulostettavan sisällön tyyppi merkkijonoksi. Ilman lainausmerkkejä, eli muodossa ”*print(Hello, World!)*”, tulostuisi komentoriville virheilmoitus. Ensimmäinen virhe tulisi huutomerkistä, koska se on Python-kielessä yksi kielen syntakseista, jolla on oma käyttötarkoituksensa. Toinen virhe tulisi puuttuvista määrityksistä *Hello*- ja *World*-pakettien osalta. (ks. Kuva 10).

```
print(Hello, World!)
```

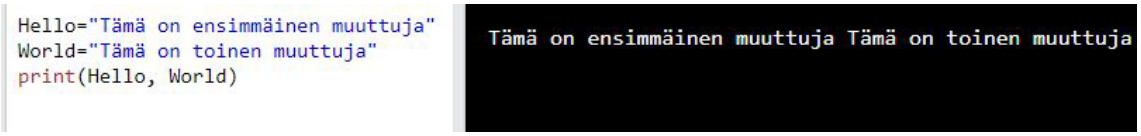
```
Traceback (most recent call last):
  File "/usr/lib/python3.8/py_compile.py", line 144, in compile
    code = loader.source_to_code(source_bytes, dfile or file,
  File "<frozen importlib._bootstrap_external>", line 846, in source_to_code
  File "<frozen importlib._bootstrap>", line 219, in _call_with_frames_removed
  File "./prog.py", line 1
    print(Hello, World!)
                      ^
SyntaxError: invalid syntax

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<string>", line 1, in <module>
  File "/usr/lib/python3.8/py_compile.py", line 150, in compile
    raise py_exc
py_compile.PyCompileError:   File "./prog.py", line 1
    print(Hello, World!)
                      ^
SyntaxError: invalid syntax
```

Kuva 10. Jätettäessä lainausmerkit pois sulkeiden sisältä, komentoriville tulostuu virheilmoitus virheellisestä syntaksista. Tässä tapauksessa se on huutomerkki.

Ilman lainausmerkkejä *print*-komento etsii *Hello*- ja *World*-nimisiä muuttujia (ks. Kuva 11). Oikein toimiessaan ohjelma tulostaisi komentoriville kahden aikaisemmin koodissa määriteltyjen *Hello*- ja *World*-muuttujien sisällön (ks. Kuva 12). Kahden siksi, että Python-kielessä pilkun pääsääntöinen tehtävä on toimia kohteiden erotinoperaattorina.



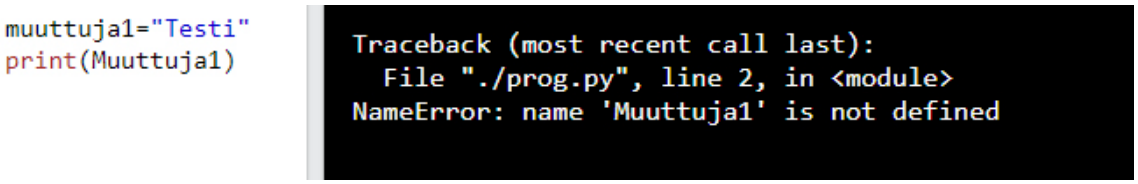
Kuva 11. Jätettäessä virheellinen syntaksi pois sulkeiden sisältä, komentoriville tulostuu uusi virheilmoitus puuttuvasta määrityksestä muuttujalle "Hello". Sama puute koskee myös "World" muuttujaa. "Hello" ja "World" ovat muuttuneet merkkijonosta kahdeksi muuttujaksi lainausmerkkien poistamisen myötä.



Kuva 12. Muuttujat "Hello" ja "World" ovat määritelty ennen tulostusta. Sisältötyyppi on kummassakin merkkijono. Nyt "print" tulostaa komentoriville muuttujiin määritellyn sisällön ilman virheilmoitusta.

"Hello, World!"-ohjelma ei kuitenkaan anna täydellistä kuvaa siitä, kuinka ohjelma tulee kirjoittaa oikein. Tarkasteltaessa kuvaa 12, voidaan jo nähdä osviittaa oikeasta kirjoitusjärjestyksestä. Tarvittavat määritykset sekä muuttujat tulee olla määriteltyinä ohjelmaan ennen kuin niitä voidaan käyttää komentojen osana. Esimerkiksi jos kuvan 12 tilanteessa muuttujat olisi määritelty tulostuskomennon jälkeen, tulostuisi komentoriville jälleen uusi virheilmoitus puuttuvista määrityksistä. Kirjoitusjärjestyksen hahmottamisen lisäksi tulee kiinnittää huomiota myös itse kirjoitusasuun. Toiminnan kannalta on myös merkitystä sillä, onko kirjoitusasussa pieniä tai isoja kirjaimia.

Esimerkkinä *”muuttuja1”* ja *”Muuttuja1”* ovat kaksi eri muuttujaa, koska ensimmäinen merkki on toisessa iso kirjain ja toisessa pieni kirjain. Jos ohjelmaan on määritetty vain *”muuttuja1”* ja jälkeenpäin käytetään koodissa *”Muuttuja1”*-nimistä muuttujaa, tulostuu komentoriville virheilmoitus määrittelemättömästä muuttujasta (ks. Kuva 13).



Kuva 13. Muuttujan kirjoitusasun ero ensimmäisen merkin kohdalla aiheuttaa virheilmoituksen puuttuvasta määrityksestä.

3.4.3 Esimerkkiohjelma: Elementtien ID-arvojen tarkistaminen

Edettäessä toiminnoiltaan laajempiin ja käytetyssä suunnittelusovelluksessa toimiviin .py-ohjelmiin, lisääntyvät komentojen, tyypitysten ja muuttujien määrät verrattuna yksinkertaisiin harjoitteisiin. Asioiden esitys- ja tapahtumajärjestys noudattaa isommissakin ohjelmissa kuitenkin samaa kaavaa kuin yksinkertaisemmissa esimerkeissä. Sivuilla 42 – 56 (kuvat 14 – 19) käyn läpi esimerkkiohjelman avulla Python-kielellä kirjoitetun ohjelman rakennetta. Ohjelma on ladattavissa Graphisoftin sivuilta nimellä *”Element ID conflict checker”* (Graphisoft, 2022). Havainnollistamisen vuoksi olen kuitenkin muokannut joidenkin muuttujien nimiä suomeksi ja selkeyttänyt ohjelman rakennetta.

```python
from archicad import ACConnection

conn = ACConnection.connect()
assert conn

acc = conn.commands
act = conn.types
acu = conn.utilities

################################ ESIASETUKSET ################################
elementit = acc.GetAllElements()
ElementinID = acu.GetBuiltInPropertyId('General_ElementID')
ElementtienArvot = acc.GetPropertyValuesOfElements(elementit, [ElementinID])

eiToistuvuutta = "Ei toistuvuutta havaittu."
Toistuvuus = ["TOISTUVUUS:", "elementillä on", "elementin ID:nä:\n"]
Toistuvuus2 = ["Vain","(yhden) elementin ID:nä on", ":\n"]

def HaeToistuvuusIlmoitus(elementIDPropertyValue, elementIds):
    return f"{Toistuvuus[0]} {len(elementIds)} {Toistuvuus[1]}
             '{elementIDPropertyValue}' {Toistuvuus[2]}"
def HaeToistuvuusIlmoitus2(elementIDPropertyValue, elementIds):
    return f"{Toistuvuus2[0]} {len(elementIds)} {Toistuvuus2[1]}
             '{elementIDPropertyValue}' {Toistuvuus2[2]}"
##############################################################################

ElementtienIdArvoDict = {}
for i in range(len(ElementtienArvot)):
    elementId = elementit[i].elementId
    ominaisuusarvo = ElementtienArvot[i].propertyValues[0].propertyValue.value
    if ominaisuusarvo not in ElementtienIdArvoDict:
        ElementtienIdArvoDict[ominaisuusarvo] = set()
    ElementtienIdArvoDict[ominaisuusarvo].add(elementId)

EiToistuvuuttalöydetty = True
for k, v in sorted(ElementtienIdArvoDict.items()):
    if len(v) > 1:
        EiToistuvuuttalöydetty = False
        print(HaeToistuvuusIlmoitus(k, v))
        print("------------------------")
    else:
        print(HaeToistuvuusIlmoitus2(k, v))
        print("------------------------")

if EiToistuvuuttalöydetty:
    print(eiToistuvuutta)
    print("------------------------")
```

Kuva 14. Valmis ohjelma.

Kuvassa 14 on nähtävissä valmis esimerkki Archicad-sovelluksessa toimivasta .py-ohjelmasta. Ohjelma on määritelty tarkistamaan Archicadissa mallinnettujen elementtien ID-arvot sekä niiden toistuvuus. Komentoriville tulostuu tieto siitä, kuinka monta kertaa yksittäinen elementin ID esiintyy projektissa. Ohjelma voidaan jakaa karkeasti kolmeen eri osaan tai sääntökokonaisuuteen, joiden tarkoitusta

käsittelin luvussa 3.2 käytetyssä IDEssä tekstien toiminnallisuutta havainnollistetaan värien avulla. Esimeriksi käyttämässäni IDEssä vaaleansiniset ovat muuttujia, keltaiset komentoja, purppurat ehtoja, tummansiniset operaattoreita ja niin edelleen. Sivulla 42 olevan kuvan 14 näkymä on peräisin Visual Studio Code IDEstä.

```
1  from archicad import ACConnection
2
3  conn = ACConnection.connect()
4  assert conn
5
6  acc = conn.commands
7  act = conn.types
8  acu = conn.utilities
9
```

Kuva 15. Ensimmäinen osa: Kytkennät.

Kuvassa 15 on ohjelman ensimmäinen osa. Kyseisessä osassa määritellään kytkennät, jonka avulla Pythonin ja Archicadin välinen rajapinta avautuu tiedonvaihtoa varten. Nämä kytkentämääritykset tulee olla kirjoitetussa ohjelmassa aina ensimmäisinä. Kytkentätyypit riippuvat käytetystä suunnitteluohjelmistosta, tarvittavista PyPi-kirjastopaketeista tai muuta rajapintaa avaavista lisäkirjastoista.

```
10 ################################ ESIASETUKSET ################################
11 elementit = acc.GetAllElements()
12 ElementinID = acu.GetBuiltInPropertyId('General_ElementID')
13 ElementtienArvot = acc.GetPropertyValuesOfElements(elementit, [ElementinID])
14
15 eiToistuvuutta = "Ei toistuvuutta havaittu."
16 Toistuvuus = ["TOISTUVUUS:", "elementillä on", "elementin ID:nä:\n"]
17 Toistuvuus2 = ["Vain","(yhden) elementin ID:nä on", ":\n"]
18
19 def HaeToistuvuusIlmoitus(elementIDPropertyValue, elementIds):
20     return f"{Toistuvuus[0]} {len(elementIds)} {Toistuvuus[1]} '{elementIDPropertyValue}' {Toistuvuus[2]}"
21 def HaeToistuvuusIlmoitus2(elementIDPropertyValue, elementIds):
22     return f"{Toistuvuus2[0]} {len(elementIds)} {Toistuvuus2[1]} '{elementIDPropertyValue}' {Toistuvuus2[2]}"
23 ##############################################################################
```

Kuva 16. Toinen osa: Määritykset / Esiasetukset.

Kuvassa 16 on nähtävillä toinen osa, jossa on määritelty tiiviisti ohjelman toimintaan tarvittavat muuttujat sekä funktiot. Muuttujia on kuusi kappaletta: *”elementit”*, *”ElementinID”*, *”Elementtiarvot”*, *”eiToistuvuutta”*, *”Toistuvuus”* ja *”Toistuvuus2”* (kaksi viimeistä on määritelty list-objektiksi hakasulkeiden avulla).

Ensimmäinen muuttuja on komento, joka hakee kaikki projektiin mallinnetut elementit, seuraava tarkkailee ID-arvoja, kolmas hakee elementtien ID-arvon *”elementit”*-muuttujan perusteella. Kolme viimeistä ovat myöhemmin koodissa käytettäviä tulostuvia tietomuuttujia. *”def”* määritystä käytetään, kun halutaan määritellä funktio. Tässä tapauksessa esimerkiksi funktio *”HaeToistuvuusIlmoitus”* yhdistää *”Toistuvuus”*-muuttujasta sekä tarkasteltavista elementeistä tietoja määritellyssä järjestyksessä. Hakasulkeiden sisällä olevat numerot viittaavat Toistuvuus-muuttujan listasisältöön. *”Toistuvuus[0]”* on merkkijono ”TOISTUVUUS:”, *”Toistuvuus[1]”* on merkkijono ”elementillä on” ja niin edelleen.

Aaltosulkeet viittaavat Python-kielessä *”dictionary”*-objektiin. *”dictionary”*-objektien tarkoitus on varastoida sen haluttu data *”key:value”*-pareiksi. Yksi pari voi olla esimerkiksi *”Malli: Ford (key), Vuosi:1964 (value)”*.

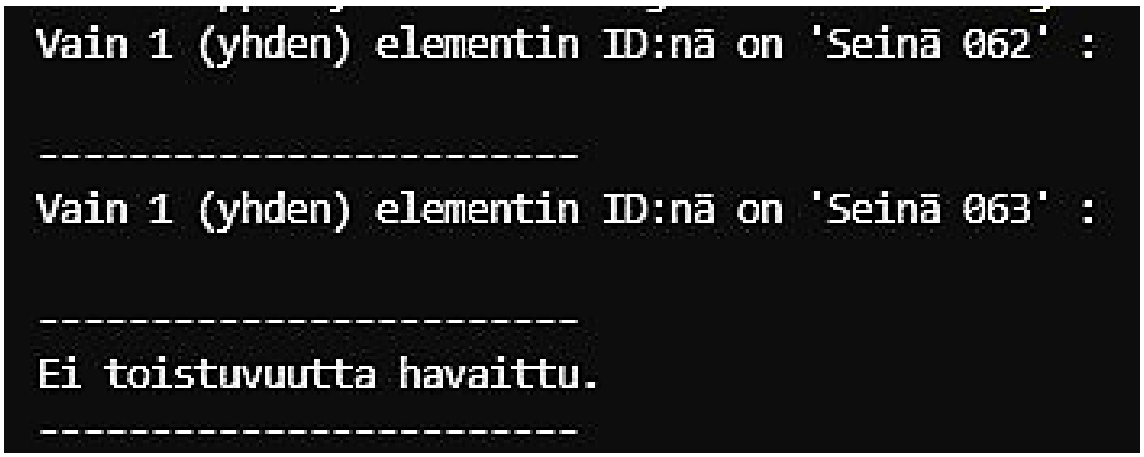
Esimerkkiohjelmassa esiintymien kokonaismäärä tulostuu *”len”*- funktion avulla kohtaan *”{len(elementIds)}”*. Tällä funktiolla haetaan toistuvien elementtien uniikki GUID-arvo ja lasketaan yhteen niiden määrät tavallisten ID-arvojen esiintymien perusteella. Tämän takia *”{len(elementIds)}”* on siis *”value”*-tyyppinen arvo. *”dictionary”*-objektia käytettäessä ei voida tulostaa komentoriville sisällöltään samaa merkkijonoa moneen kertaan, joten tämän takia toistuva ID-arvo saadaan tulostumaan merkkijonona vain kerran kohtaan *”{elementIDPropertyValue}”* ja on täten *”key”*-tyyppinen arvo. Funktiota *”HaeToistuvuusIlmoitus2”* käytetään vain niiden elementtien ID-arvon tulostamiseen, jotka sisältävät projektissa vain kerran esiintyvän ID:n. Funktioita kutsutaan tulostumaan osassa 3, jolloin varsinaiset tarkastustiedot sijoitetaan funktioissa ennalta määriteltyihin väleihin.

```
24
25  ElementtienIdArvoDict = {}
26  for i in range(len(ElementtienArvot)):
27      elementId = elementit[i].elementId
28      ominaisuusarvo = ElementtienArvot[i].propertyValues[0].propertyValue.value
29      if ominaisuusarvo not in ElementtienIdArvoDict:
30          ElementtienIdArvoDict[ominaisuusarvo] = set()
31      ElementtienIdArvoDict[ominaisuusarvo].add(elementId)
32
33  EiToistuvuuttalöydetty = True
34  for k, v in sorted(ElementtienIdArvoDict.items()):
35      if len(v) > 1:
36          EiToistuvuuttalöydetty = False
37          print(HaeToistuvuusIlmoitus(k, v))
38          print("------------------------")
39      else:
40          print(HaeToistuvuusIlmoitus2(k, v))
41          print("------------------------")
42
43  if EiToistuvuuttalöydetty:
44      print(eiToistuvuutta)
45      print("------------------------")
```

Kuva 17. Kolmas osa: Toiminta.

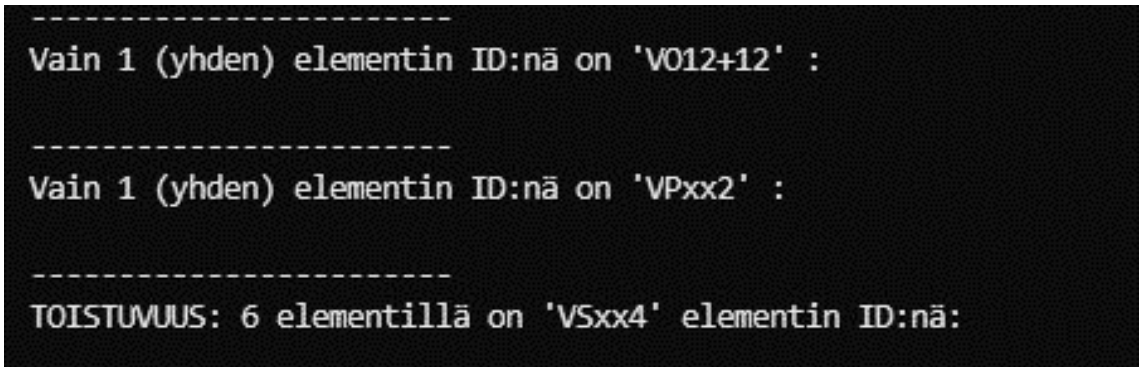
Kuvassa 17 on nähtävillä ohjelman kolmas osa, joka hyödyntää esiasetuksessa määriteltyjä muuttujia ja loppuosa koodista käy läpi elementit sekä määrittää tulosteiden sisällön ”*if*”-ehtojen kautta. Riveillä 25 – 31 määritellään ID-arvojen yhteenlaskutapahtuma. Tässä osassa tapahtuu ID-arvojen muuttaminen merkkijonoista numeeriseen muotoon, jotta yhteenlaskusuoritus onnistuu ja tulostuu oikein. Riveillä 33 – 45 koodi ohjaa tulostumaan komentoriville joko ”*HaeToistuvuusIlmoitus*”, ”*HaeToistuvuusIlmoitus2*” tai ”*eiToistuvuutta*” riippuen ID-arvon esiintymismäärästä projektissa.

Lopputuloksena komentoriville ilmestyy tieto erilaisten ID-arvojen toistuvuuskerroista projektiin sijoitetuissa elementeissä. (ks. Kuva 18). Jos tarkastuksen läpikäyneistä elementeistä ei löydy samaa ID-arvoa vähintään kahteen kertaan, tulostuu lopuksi muuttujaan *”eiToistuvuutta”* sisällöksi määritelty merkkijono ”Ei toistuvuutta havaittu”.



Kuva 18. Tuloste komentorivillä.

Jos koodi löytää saman ID:n useammasta elementistä, komentoriville tulee ilmoitus elementtien määrästä sekä kyseisten elementtien ID-arvosta (ks. Kuva 19).



Kuva 19. Tuloste kun projektin elementeistä löytyy toistuvia ID-arvoja.

Esimerkkiohjelma ei ole sidottu pelkästään ID-arvojen tarkastamiseen, vaan sisältöä muokkaamalla sama ohjelma voidaan muuttaa toimimaan toisen tarkastusta vaativan asian tarkastajana.

4 CASE-OHJELMAT

Tässä luvussa käsittelen ohjelmien kirjoittamista case-tapausten kautta. Kirjoitin jokaiseen tapaukseen toiminnaltaan erilaisen ohjelman. Asetin ennen ohjelmien kirjoittamisen aloittamista niille seuraavan vaatimuksen: ohjelmien tulisi olla toimivia, jatkokehitettävissä sekä muuntojoustavia muita projekteja silmällä pitäen. Päätavoitteenani oli siis luoda case-tapausten ohjelmien avulla runko, jota voisin hyödyntää helposti tulevaisuudessa kirjoittaessani uusia ohjelmia.

Hyödynsin ohjelmien kirjoittamiseen myös muiden tekijöiden kirjoittamia lisäosia tai kirjastopaketteja, jotka lisäsivät Python-kielen toimintamahdollisuuksia sekä avasivat kielelle rajapintaa Archicad-työympäristössä. Laajempien käyttömahdollisuuksien turvin pystyin tarkastelemaan paremmin ohjelmointitaidon soveltuvuutta arkkitehdin työvälineenä, sekä kielen opetteluun vaadittua ajallista panostusta. Mainitsen käytetyt lisäosat erikseen kunkin case-tapauksen yhteydessä.

Ennen kuin aloitin ohjelmien kirjoittamisen case-tapauksia varten, kävin läpi alkuvalmistelut, jotka tulee tehdä Pythonilla kirjoitettujen ohjelmien ajamiseen Archicadissa. Näitä alkuvalmisteluja ovat Pythonin sekä tarvittavien kirjastopakettien asentaminen, kytkentöjen aktivoiminen sekä IDEn asennus. IDEnä ohjelmien kirjoittamiseen käytin Microsoft Visual Studiota Code -ohjelmaa. Kun olin aktivoinut Python-tulkit sekä -kielen toimintaan suunnittelusovelluksen asetuksissa sekä käytetyssä IDEssä, pystyin ajamaan valmiit ohjelmat IDEstä suoraan Archicadiin. Hyödynsin IDEä myös kirjoittamieni ohjelmien virheentarkistuksessa sekä testiajoissa ennen ohjelmassa asetettujen määritysten varsinaista ajoa Archicadissa oleviin elementteihin.

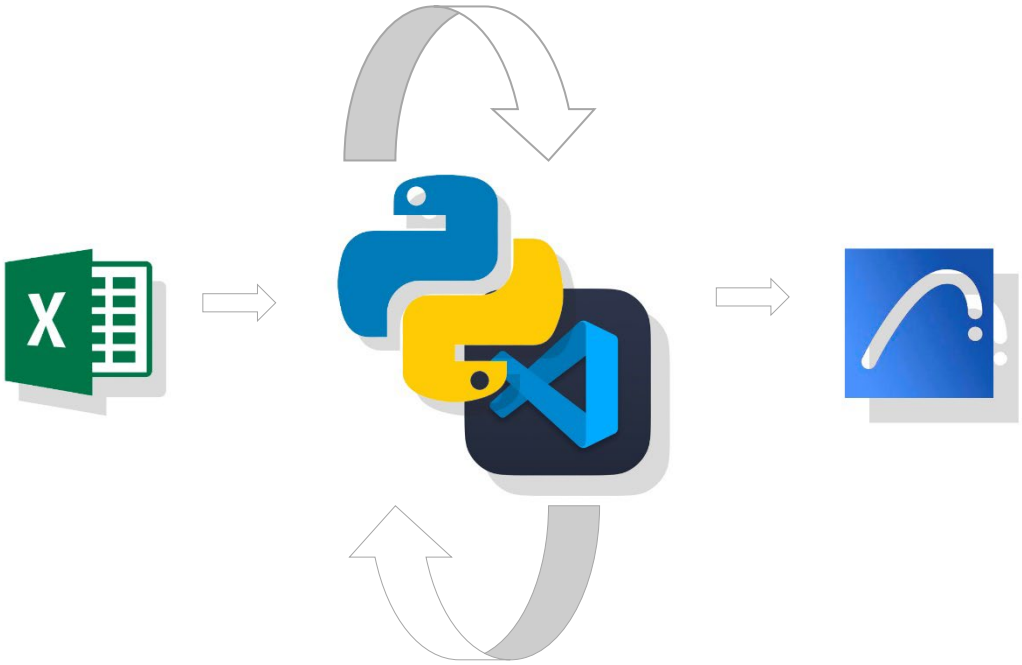
Ensimmäisessä case-tapauksessa kirjoitin ohjelman elementtien luokitustiedon ajamiseen yhdellä kertaa kaikkiin projektissa ilmeneviin elementtityyppeihin. Seuraavassa case-tapauksessa kirjoitin ohjelman tarkastelemaan tilojen pinta-alojen prosentuaalista osuutta suhteutettuna tarkasteltavan kerroksen tila-aloihin. Projektista haetun datan avulla ohjelma loi automaattisia graafisia esityksiä. Kolmannessa case-tapauksessa kirjoitin ohjelman väestönsuojalaskelmaa varten.

Ohjelma haki tarvittavat tiedot projektiin mallinnetuista vyöhyke-elementeistä.

Muita käyttäjiä varten päädyin jakamaan ohjelmien sisällön osiin. Osituksen tarkoitus oli helpottaa ohjelmien rakenteen hahmottamista sekä osoittamaan kohdat, joihin tulee tehdä tarpeen vaatiessa muutoksia tai lisäyksiä. Tarkastelin Case-ohjelmien rakennetta myös siitä näkökulmasta, kuinka tiiviisti ohjelma on järkevää kirjoittaa, jotta ohjelman koodin muokkaus onnistuisi helposti mahdollisimman monelta käyttäjältä. Tämä johti siihen, että rivimäärät sekä toisto ohjelmissa kasvoivat, mutta vastaavasti ohjelman sisällön ymmärrettävyys oli parempi. Näin ollen ohjelmaa voisivat muokata myös sellaiset henkilöt, joilla ei ole aikaisempaa ohjelmointikokemusta. Koodit olisivat siis olleet mahdollista toteuttaa tiiviimpinä, mutta tällöin sisällön muokkaaminen olisi haastavampaa kokemattomammille käyttäjille.

Ohjelmien kirjoittamiseen vaaditun osaamistason saavuttamiseen minulta kului noin neljä kuukautta enkä lähtötilanteessa omannut aikaisempaa kokemusta Python-kielen kirjoittamisesta. Tämän neljän kuukauden aikana opiskelin satunnaisina hetkinä Python-kielen teoriaa ja tein erilaisia ohjelmointiin liittyviä harjoitustehtäviä. Ensimmäiset sekä toiminnaltaan yksinkertaisemmat versiot ohjelmista oli mahdollista kirjoittaa jo kahden kuukauden itseopiskelun jälkeen.

4.1 Case 1: Projektin elementtien luokittelu



Case-tapauksen projektissa tehtävänäni oli määritellä mallinnettuihin Archicad-elementteihin luokittelut käytetyn tasojärjestelmän mukaisesti. Kirjoitin Python-ohjelman suorittamaan luokittelun automaattisesti perustuen mallinnettujen elementtien tasoihin sekä tarvittaessa niihin liittyviin tunnistetietoihin. Ohjelma noutaa Python-komentojen perusteella erillisen Excel-taulukon, jossa vierekkäisissä sarakkeissa ovat taso- ja tasoa vastaava luokitusnumero. Näin lähtötiedot muuttuvat ohjelman ajovaiheessa rivityksen perusteella *”key:value”* -pareiksi. Lisäsin koodiin poikkeukset niille elementeille, jotka ovat osa toista elementtiä kuten ikkunat ja ovet. Tällaiset elementit menisivät muutoin isäntäelementin tason mukaan toisen luokitusnumeron alle. Ajot Archicadin elementteihin tein Visual Studio Code -ohjelmassa.

Archicadilla toteutetuissa tietomalliprojekteissa elementtien luokittelu on yksi tärkeimmistä tehtävistä. Luokat eivät ainoastaan avaa ennalta tehtyjä ominaisuuksia elementeille, vaan luokitukset myös määrittävät elementtien tyypityksen IFC-tallenteessa. Esimerkiksi ikkunatyökalulla mallinnettu elementti voi näyttää muodoltaan ikkunalta, mutta luokituksen perusteella se voidaan määrätä kääntymään vaikkapa oveksi, seinäksi tai kiintokalusteeksi. Ilman luokitteluja elementit kääntyvät Proxy-objekteiksi, jotka eivät sisällä IFC-tallenteessa tietoa niiden varsinaisesta käyttötarkoituksesta.

Tähän case-tapaukseen kirjoittamaani ohjelmaa on mahdollista hyödyntää kaikissa projekteissa, joissa on käytössä sama luokitus- ja tasojärjestelmä. Ohjelma vaatii toimiakseen Archicadin Python-kirjastopaketin lisäksi Openpyxl-kirjaston. Openpyxl on tarkoitettu avaamaan rajapinta tiedon lukemiseen tallennetusta Excel-taulukosta sekä tarvittaessa Archicad-projektissa olevien elementtien tietojen kirjoittamiseen suoraan Excel-taulukkoon.

Lopullisen ohjelmaversion kirjoittaminen testiajoineen sekä virheentarkistuksineen kesti noin yhden työpäivän verran. Kirjoittamisen kestossa on myös huomioituna erillisen Excel-taulukon rakentaminen.

```
#_1_############################ KYTKENNÄT ############################


from archicad import ACConnection
from openpyxl import Workbook,load_workbook
import archicad
import sys
import re
from archicad.releases.ac25.b3000types import ElementId, ElementPropertyValue
conn = ACConnection.connect()
assert conn
acc = conn.commands
act = conn.types
acu = conn.utilities

#_2_############################ ALKUMÄÄRITYKSET ############################

AllElements = acc.GetAllElements()
LibraryPart=acu.GetBuiltInPropertyId('General_LibraryPartName')
Layer = acu.GetBuiltInPropertyId('ModelView_LayerName')
elementType=acu.GetBuiltInPropertyId('General_Type')
value =acc.GetPropertyValuesOfElements(AllElements,[Layer])
value2=acc.GetPropertyValuesOfElements(AllElements,[elementType])
value3=acc.GetPropertyValuesOfElements(AllElements,[LibraryPart])
class1 = acu.FindClassificationSystem('#Luokitusjärjestelmä#')
elementClass1=acc.GetClassificationsOfElements(AllElements,[class1])
elementClass2=acc.GetAllClassificationsInSystem(class1)


#_3_############################ EXCEL-TIEDOSTON HAKU ############################

wb = load_workbook('#Sijainti\Tiedosto.xlsx#')
ws = wb.active

def iter_rows(ws):
    for row in ws.iter_rows():
        yield[cell.value for cell in row]
res = iter_rows(ws)
mydict=dict(res)
my_list2=[mydict]
```

Kuva 20. Osat 1–3.

Kuvassa 20 ovat ohjelman osat 1 – 3. Nämä kolme ensimmäistä osaa sisältävät toiminnan kannalta vaaditut kytkennät kirjastoihin sekä Archicadiin, muuttujien määritykset sekä hakukomennon halutulle Excel-taulukkotiedostolle. Sulkeiden sisällä olevat *”#Luokitusjärjestelmä”*, *”#Luokka”* sekä *”#Taso”* viittaavat projektissa käytettyihin taso- ja luokitusjärjestelmiin. Edellä mainitut kohdat tulee korvata projektissa käytettyjen luokitus- sekä tasojärjestelmien mukaisiksi, jotta ohjelma toimii tarkoitetulla tavalla.

```
#_4_########################## LUOKITTELUAJO ############################

elem_classes = []
for index,element2 in enumerate(AllElements):
    Tasotieto=str(value[index].propertyValues[0].propertyValue.value)
    Elementtityyppi=str(value2[index].propertyValues[0].propertyValue.value)
    ObjektiNimi=str(value3[index].propertyValues[0].propertyValue)

    ###_4.0_###_YLEISET_####
    if "\x14"==Tasotieto:
        class2 = acu.FindClassificationItemInSystem('#Luokitusjärjestelmä#','#Luokka#')
        class_id = act.ClassificationId(class1,class2.classificationItemId)
        elem_class = act.ElementClassification(element2.elementId,class_id)
        print("#Luokka#",Elementtityyppi,Tasotieto)
    if (
            "\x14"!=Tasotieto and
            "Ikkuna"!=Elementtityyppi and
            "Ovi"!=Elementtityyppi and
            "Aukko"!=Elementtityyppi and
            "Kattoikkuna"!=Elementtityyppi
        ):
        values_of_tasotieto=(str([a_dict[Tasotieto] for a_dict in my_list2])[2:-2])
        luokka=str(values_of_tasotieto)
        class2 = acu.FindClassificationItemInSystem('#Luokitusjärjestelmä#',values_of_tasotieto)
        class_id = act.ClassificationId(class1,class2.classificationItemId)
        elem_class = act.ElementClassification(element2.elementId,class_id)
        print(luokka,Elementtityyppi,Tasotieto)

    ###_4.1_###_Ikkunat_####
    if "Ikkuna"==Elementtityyppi:
        if ("#Taso#"==Tasotieto and "Ikkuna PK21" in ObjektiNimi):
            class2 = acu.FindClassificationItemInSystem('#Luokitusjärjestelmä#','#Luokka#')
            class_id = act.ClassificationId(class1,class2.classificationItemId)
            elem_class = act.ElementClassification(element2.elementId,class_id)
            print("#Luokka#", Elementtityyppi,Tasotieto)
        elif ("#Taso2"==Tasotieto and "Ikkuna PK21" in ObjektiNimi):
            class2 = acu.FindClassificationItemInSystem('#Luokitusjärjestelmä#','#Luokka#')
            class_id = act.ClassificationId(class1,class2.classificationItemId)
            elem_class = act.ElementClassification(element2.elementId,class_id)
            print("#Luokka#", Elementtityyppi,Tasotieto)
        elif ("#Taso#"==Tasotieto and "Pinta-alue PK21" in ObjektiNimi):
            ObjektiNimi=str(value3[index].propertyValues[0].propertyValue.value)
            class2 = acu.FindClassificationItemInSystem('#Luokitusjärjestelmä#','#Luokka#')
            class_id = act.ClassificationId(class1,class2.classificationItemId)
            elem_class = act.ElementClassification(element2.elementId,class_id)
            print("#Luokka#", Elementtityyppi,ObjektiNimi,Tasotieto)

    ###_4.2_###_OVET_####
    if "Ovi"==Elementtityyppi:
        if ("#Taso#"==Tasotieto):
            class2 = acu.FindClassificationItemInSystem('#Luokitusjärjestelmä#','#Luokka#')
            class_id = act.ClassificationId(class1,class2.classificationItemId)
            elem_class = act.ElementClassification(element2.elementId,class_id)
            print("#Luokka#", Elementtityyppi,Tasotieto)
        elif ("#Taso2"==Tasotieto):
            class2 = acu.FindClassificationItemInSystem('#Luokitusjärjestelmä#','#Luokka#')
            class_id = act.ClassificationId(class1,class2.classificationItemId)
            elem_class = act.ElementClassification(element2.elementId,class_id)
            print("#Luokka#", Elementtityyppi,Tasotieto)
```

Kuva 21. Aliosa 4.0–4.2.

```
###_4.3_###_AUKOT_####
    if "Aukko"==Elementtityyppi:
        if ("#Taso3#" in Tasotieto):
            class2 = acu.FindClassificationItemInSystem('#Luokitusjärjestelmä#','#Luokka#')
            class_id = act.ClassificationId(class1,class2.classificationItemId)
            elem_class = act.ElementClassification(element2.elementId,class_id)
            print("#Luokka#", Elementtityyppi,Tasotieto)
        elif ("#Taso4#" in Tasotieto):
            class2 = acu.FindClassificationItemInSystem('#Luokitusjärjestelmä#','#Luokka#')
            class_id = act.ClassificationId(class1,class2.classificationItemId)
            elem_class = act.ElementClassification(element2.elementId,class_id)
            print("#Luokka#", Elementtityyppi,Tasotieto)

###_4.4_###_Kattoikkunat_####
    if "Kattoikkuna"==Elementtityyppi:
        if ("#Taso3#" in Tasotieto):
            class2 = acu.FindClassificationItemInSystem('#Luokitusjärjestelmä#','#Luokka#')
            class_id = act.ClassificationId(class1,class2.classificationItemId)
            elem_class = act.ElementClassification(element2.elementId,class_id)
            print("#Luokka", Elementtityyppi,Tasotieto)

###_4.x_###_Lisää uusi aliosa tämän otsakkeen jälkeen_####


    elem_classes.append(elem_class)
    acc.SetClassificationsOfElements(elem_classes)

print("Luokitukset ajettu elementteihin. Luokittele manuaalisesti elementtien sisäiset
elementtiosat (Verhorakenne, kaide, porras)")
#_5_########################## OHJELMA PÄÄTTYY ############################
```

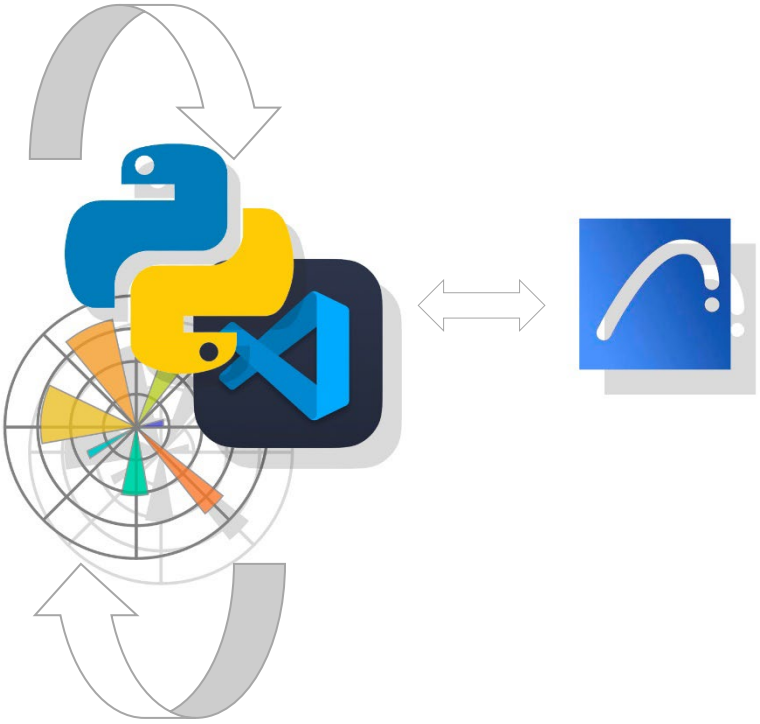
Kuva 22. Aliosat 4.3–4.x, Osa 5.

Kuvissa 21 ja 22 ovat osa 4(aliosat 4.0 - 4.x) sekä osa 5. Aliosat koskevat ikkuna-, ovi-, aukko-, ja kattoikkunaelementtejä eli sellaisia elementtityyppejä, jotka mallinnetaan osaksi isäntäelementtiä kuten seinää. Ilman poikkeutusta, tällaiset elementit saisivat luokitusnumeron niiden isäntäelementtien mukaan. Kirjoitin ohjelman niin, että aliosia voidaan lisätä tai poistaa tarpeen mukaan. Lisääminen tapahtuu kopioimalla koodissa oleva aliosapaketti kohtaan *”4.x”*, tekemällä siihen tarvittavat sisältömuutokset ja lisäämällä poikkeus kyseisestä elementtityypistä aliosaan 4.0. Ensimmäisessä aliosassa (ks. kuva 21, kohta *”4.0 Yleiset”*) pois suljettavat kohteet merkitään *”!=”*- tai *”not in”* -operaattorilla. *”!=”* vaatii, että pois suljettavan asian sisältö on tismalleen sama kuin muuttujan sisältö, johon verrataan. *”not in”* operaattoria käytettäessä riittää, että kyseinen osa löytyy vertailtavasta muuttujasta. Lopuksi ohjelma tulostaa tulosteen merkitsemään ajon päättymistä.

Yleensä elementtien luokittelu tapahtuu projektissa samaa tahtia kuin itse mallinnustyö. Tällöin elementtien luokittelu on pieni osa suuremmassa suunnittelukokonaisuudessa. Luokittelun tekeminen silloin kun mallinnus ja suunnitelmat ovat jo pitkällä, voi olla isompaa ajallista panostusta vaativa kokonaisuus. Vasta tuolloin tapahtuvaan luokitustietojen määrittelyyn olisi muutamia vaihtoehtoja, kuten Archicadin elementtitaulukkojen hyödyntäminen sekä etsi-työkalu. Kirjoittamani ohjelma kuitenkin mahdollisti, että nämä manuaalista työtä vaativat avut voitiin jättää pois ja pystyin suorittamaan elementtien luokittelut automaation avulla tarkasti ja nopeasti.

Palasin jatkokehittämään ohjelmaa muutama kuukausi sen varsinaisen valmistumisen jälkeen. Lukiessani viestiketjuja Python-ohjelmointiin liittyvältä foorumilta, löysin sieltä ratkaisutavan, jota käyttämällä ohjelmakoodi olisi mahdollista ajaa yhdellä kertaa useampaan yhtä aikaa avoinna olevaan projektitiedostoon. Ratkaisun toimintaperiaate liittyy porttiosoitteiden käyttämiseen. Lisäämällä porttiosoitteet koodiin, sain ohjelman ajamaan luokittelut jopa kymmeneen projektiin yhtäaikaisesti. Jokainen avoinna oleva Archicad-instanssi varaa Python-yhteyttä varten käyttöönsä yksilöllisen porttiosoitteen portista 19723 alkaen. Ilman erillistä osoitemääritystä koodissa, porttiosoite ajoa varten on aina vakio eli 19723.

4.2 Case 2: Tilojen jakaumien esittäminen graafisesti



Case-tapauksen projektissa tehtävänäni oli tuottaa graafiset diagrammiesitykset kerrosten eri vaihtoehtoratkaisuista. Näistä graafeista tuli käydä ilmi eri vaihtoehdoissa esiintyvien tilatyyppien viemä prosentuaalinen osuus koko kerroksen pinta-alasta. Kirjoitin tätä varten Python-ohjelman, joka tarkastaa Archicadiin mallinnettujen vyöhyke-elementtien nimet ja tilatyypit niputtaen samanlaiset nimi- ja tyyppiarvot omaavat vyöhyke-elementit yhdeksi kokonaisuudeksi graafia varten. Tämän jälkeen ohjelma laskee vaihtoehdoittain samantyyppisten vyöhykkeiden pinta-alat yhteen ja vertaa tulosta kaikkien huonealojen summaan kyseisessä kerroksessa. Ohjelma vaatii toimiakseen Archicadin Python-kirjastopaketin lisäksi Matplotlib-kirjaston. Kirjasto mahdollistaa projektista luettavan tiedon muunnoksen graafiseen muotoon.

Projektien luonnosvaiheissa tarkastellaan yleensä rinnakkain erilaisia ratkaisuvaihtoehtoja. Nämä vaihtoehdot esitetään asiakkaalle pohjapiirustusten ja pinta-alalukujen avulla. Pohjapiirustuksissa olevat tilat ovat usein korostettuina eri värein, joilla asiakkaalle havainnollistetaan tilan käyttötarkoitus, tilojen sijoittuminen sekä muoto.

Pelkät pohjat sekä pinta-alat eivät välttämättä kuitenkaan havainnollista tarpeeksi selkeästi sitä kuinka paljon vaihtoehdoissa esiintyvät tilatyypit vievät prosentuaalisesti tilaa suhteutettuna projektin kokonaisalaan. Tällaiselle kokoavalle esitystavalle voi syntyä tarve tapauksissa, joissa esiteltäviä vaihtoehtoja on useita ja niissä esiintyvien tilojen suhteellista alaa halutaan verrata keskenään. Data voidaan näissä tilanteissa tiivistää selkeään esitysmuotoon, joita voivat olla esimerkiksi ympyrä-, piste- sekä pylväsdiagrammit, joilla kaikki vaihtoehdot sekä niiden suhde toisiinsa voidaan esittää yhdellä arkilla.

Rungon kirjoittamiseen aikaa minulta kului noin puoli työpäivää ja lopullisen ohjelmaversion kirjoittaminen testiajoineen sekä virheentarkistuksineen kesti noin kolme työpäivää. Aikaa vievin osuus kirjoituksessa oli löytää koodin oikea kirjoitusjärjestys diagrammien luontia sekä esitystapaa varten. Ajoin testiajot ja valmiin ohjelman Visual Studio Code IDEssä.

```
#_1_########################## KYTKENNÄT ##########################


from operator import contains
from os import remove
from archicad import ACConnection
from mpl_toolkits.mplot3d import Axes3D
import archicad
import sys
import re
import numpy as np
from optparse import Values
from archicad import ACConnection
from matplotlib import collections, projections
import matplotlib.pyplot as plt
import collections
from collections import Counter
from collections import defaultdict
conn = ACConnection.connect()
assert conn


from archicad.releases.ac25.b3000types import ElementId, EnumValueId
conn = ACConnection.connect()

assert conn

acc = conn.commands
act = conn.types
acu = conn.utilities

#_2_########################## ALKUMÄÄRITYKSET ##########################

zones=acc.GetElementsByType('Zone')
Type=acu.GetBuiltInPropertyId('General_Type')
Area=acu.GetBuiltInPropertyId('Zone_MeasuredArea')
ZoneName=acu.GetBuiltInPropertyId('Zone_ZoneName')
Layer=acu.GetBuiltInPropertyId('Modelview_LayerName')
Phase=acu.GetBuiltInPropertyId('#Vaiheistus#')
ProjectName=acu.GetUserDefinedPropertyId('#OminaisuusRyhmä#','#Projektin nimi -ominaisuus')
StoryName=acu.GetUserDefinedPropertyId('#OminaisuusRyhmä#','#Kerrosominaisuus')
Areavalue=acc.GetPropertyValuesOfElements(zones,[Area])
ZoneNameValue=acc.GetPropertyValuesOfElements(zones,[ZoneName])
ZoneLayerValue=acc.GetPropertyValuesOfElements(zones,[Layer])
boundingboxes=acc.Get3DBoundingBoxes(zones)
ProjectNameValue=acc.GetPropertyValuesOfElements(zones,[ProjectName])
StoryNameValue=acc.GetPropertyValuesOfElements(zones,[StoryName])
PhaseValue=acc.GetPropertyValuesOfElements(zones,[Phase])
counter = collections.Counter()
```

Kuva 23. Osat 1–2.

Kuvassa 23 on esitetty ohjelman kaksi ensimmäistä osaa, jotka sisältävät kytkennät kirjastoihin ja Archicadiin sekä muuttujien määritykset kuten ensimmäisessä case-tapauksessa. Osassa kaksi määritykset ovat kuitenkin erilaiset verrattuna ensimmäisen case-tapauksen ohjelmaan, koska kaikki haettavat asiat eivät ole samoja. Oranssilla fonttivärillä korostetut sekä ”*#*”-etu- ja loppusyntaksilla merkityt kentät tarkoittavat, että näissä kohdissa haetaan muuttujan parametriksi projektikohtaista

ominaisuutta, ominaisuusryhmää tai tasoa, joka ei välttämättä toistu saman nimisenä projektista toiseen. Päädyin case-tapausten kuvissa tähän esitystapaan, jotta niitä ei tämän diplomityön sivuilla sekoitettaisi ennalta määrättyihin parametrien nimiin (kuten ”*Zone_MeasuredArea*”), joita ei voi nimetä itse. Koodissa edellä mainitun kaltaiset kiinteiden parametrien arvot haetaan Archicad-projektista osaksi muuttujaa esimerkiksi komennolla ”*acu.GetBuiltInPropertyId*”*(BuiltIn = sisäänrakennettu).*

Varsinaisessa .py-tiedostoksi tallennetussa ohjelmassa kirjoitin itse nimettävät parametrikentät vastaamaan case-tapauksen projektissa käytettyjä ominaisuusnimikkeitä. Tällaisten parametrien data haetaan Archicad-projektista komennolla ”*acu.GetUserDefinedPropertyId*” *(UserDefined = käyttäjän määrittelemä).* Nämä kentät voidaan muuttaa tarpeen vaatiessa vastaamaan Archicad-projektissa olevaa toisen nimistä ominaisuutta tai ominaisuusryhmää. Esimerkiksi kerrostiedon ominaisuus voi sijaita ominaisuusryhmässä ”*Rakennusdata*” nimellä ”*Kerrosnimi*”. Tällöin muuttujan ”*StoryName*” parametrikentät ”’*#Ominaisuusryhmä#*’, ’*#Kerrosominaisuus*’” kirjoitettaisiin ohjelmakoodissa muotoon ”’*Rakennusdata*’, ’*Kerrosnimi*’”.

```
#_3_######## TILOJEN PINTA-ALOJEN YHTEENLASKU JA RAJAUKSET BOUNDINGBOX ##########


AreaArray1=[]
AreaArray2=[]
AreaArray3=[]
for index,element in enumerate(zones):
    element = element.elementId
    ominaisuusarvo2 = round((float(Areavalue[index].propertyValues[0].propertyValue.value)),2)
    Tasotieto= str(ZoneLayerValue[index].propertyValues[0].propertyValue.value)
    Vyöhyketieto=str(ZoneNameValue[index].propertyValues[0].propertyValue.value)
    Kerrostieto=str(StoryNameValue[index].propertyValues[0].propertyValue.value)
    Versiotieto=str(PhaseValue[index].propertyValues[0].propertyValue.value)
    Boksit_yMin=boundingboxes[index].boundingBox3D.yMin
    Boksit_xMin=boundingboxes[index].boundingBox3D.xMin
    Boksit_yMax=boundingboxes[index].boundingBox3D.yMax
    Boksit_xMax=boundingboxes[index].boundingBox3D.xMax
    Boksirajat=(((Boksit_xMax<=#xMax_metreinä#)
                and(Boksit_yMax<=#yMax_metreinä#))
                and((Boksit_xMin>=#xMin_metreinä#)
                and(Boksit_yMin>=#yMin_metreinä#)))
    if ('#Tilataso#'==Tasotieto and ('VE1' in Versiotieto) and '#Kerros#'==Kerrostieto
        and Boksirajat==True):
        AreaArray1.append(ominaisuusarvo2)
        print(Vyöhyketieto, Versiotieto, Kerrostieto)
    if ('#Tilataso#'==Tasotieto and ('VE2' in Versiotieto) and '#Kerros#'==Kerrostieto
        and Boksirajat==True):
        AreaArray2.append(ominaisuusarvo2)
        print(Vyöhyketieto, Versiotieto, Kerrostieto)
    if ('#Tilataso#'==Tasotieto and ('VE3' in Versiotieto) and '#Kerros#'==Kerrostieto
        and Boksirajat==True):
        AreaArray3.append(ominaisuusarvo2)
        print(Vyöhyketieto, Versiotieto, Kerrostieto)
    else:
        remove
Summa1=sum(AreaArray1)
Summa2=sum(AreaArray2)
Summa3=sum(AreaArray3)
print("------------------------")
print("VE1 Alojen summa on:", round((Summa1),2))
print("VE2 Alojen summa on:", round((Summa2),2))
print("VE3 Alojen summa on:", round((Summa3),2))
print("------------------------")
```

Kuva 24. Osa 3.

Kuvassa 24 on osa 3, jossa määritellään tarkasteltavan alueen rajat (*”boundingBox3D”*). Ohjelma etsii näiden rajaavien muotojen sisältä vyöhyke-elementtejä, jotka ovat mallinnettu tietylle tasolle, tiettyyn kerrokseen sekä esiintyvät vain tietyssä versiossa. Näiden ehtojen perusteella ohjelma luo kolme listaa, johon ehdot täyttävien vyöhyke-elementtien pinta-alat lasketaan yhteen versioittain kahden desimaalin tarkkuudella. Summat siirretään muistiin muuttujiin *”Summa1”*, *”Summa2”* ja *”Summa3”*, jotta pinta-aloista saatu data voidaan käyttää hyödyksi ohjelman myöhemmässä vaiheessa luotavissa ympyrädiagrammeissa.

```
#_4_###### TILATYYPIT DICTIONARY-OBJEKTIKSI, JOSSA NIMET JA ALAT (KEY:VALUE) PAREJA ########

#_4.1_########################## VERSIO 1 TARKASTELU ##########################
AreaArrayDict1=[]
for index,element in enumerate(zones):
    element = element.elementId
    Pinta_ala=round((float(Areavalue[index].propertyValues[0].propertyValue.value)),2)
    Tasotieto= str(ZoneLayerValue[index].propertyValues[0].propertyValue.value)
    Vyöhyketieto=str(ZoneNameValue[index].propertyValues[0].propertyValue.value)
    Projektitieto=str(ProjectNameValue[index].propertyValues[0].propertyValue.value)
    Kerrostieto=str(StoryNameValue[index].propertyValues[0].propertyValue.value)
    Versiotieto=str(PhaseValue[index].propertyValues[0].propertyValue.value)
    if ('#Tilataso#'==Tasotieto and ('VE1' in Versiotieto) and '#Kerros#'==Kerrostieto
        and Boksirajat==True):
        ZoneDict1={Vyöhyketieto:Pinta_ala}
        AreaArrayDict1.append(ZoneDict1)
        print("Ominaisuusarvo ZoneDict1:",ZoneDict1,"Ymin:",Boksit_yMin,"Ymax:",Boksit_yMax,
              "Xmin:", Boksit_xMin,"Xmax:",Boksit_xMax)
    else:
        remove
print("------------------------")
print("AreaArrayDict1-listan sisältö:",AreaArrayDict1)
print("------------------------")
for d1 in AreaArrayDict1:
    counter.update(d1)
result1 = dict(counter)
print("Dictionary1-objektin sisältö:", str(counter))
values1, counts1 = zip(*counter.most_common())
print("------------------------")
print("Esiintyvät tilanimet:",values1)
print("Pinta-alat (saman nimiset yhdistetty):", counts1)
counter.clear()
print("------------------------")

#_4.2_########################## VERSIO 2 TARKASTELU ##########################
AreaArrayDict2=[]
for index,element in enumerate(zones):
    element = element.elementId
    Pinta_ala=round((float(Areavalue[index].propertyValues[0].propertyValue.value)),2)
    Tasotieto= str(ZoneLayerValue[index].propertyValues[0].propertyValue.value)
    Vyöhyketieto=str(ZoneNameValue[index].propertyValues[0].propertyValue.value)
    Projektitieto=str(ProjectNameValue[index].propertyValues[0].propertyValue.value)
    Kerrostieto=str(StoryNameValue[index].propertyValues[0].propertyValue.value)
    Versiotieto=str(PhaseValue[index].propertyValues[0].propertyValue.value)
    if ('#Tilataso#'==Tasotieto and ('VE2' in Versiotieto) and '#Kerros#'==Kerrostieto
        and Boksirajat==True):
        ZoneDict2={Vyöhyketieto:Pinta_ala}
        AreaArrayDict2.append(ZoneDict2)
        print("Ominaisuusarvo ZoneDict2:",ZoneDict2,"Ymin:",Boksit_yMin,"Ymax:",Boksit_yMax,
              "Xmin:", Boksit_xMin,"Xmax:",Boksit_xMax)
    else:
        remove
print("------------------------")
print("AreaArrayDict2-listan sisältö:",AreaArrayDict2)
print("------------------------")
for d2 in AreaArrayDict2:
    counter.update(d2)
result2 = dict(counter)
print("Dictionary2-objektin sisältö:", str(counter))
values2, counts2 = zip(*counter.most_common())
print("------------------------")
print("Esiintyvät tilanimet:",values2)
print("Pinta-alat (saman nimiset yhdistetty):", counts2)
counter.clear()
print("------------------------")
```

Kuva 25. Aliosat 4.1–4.2.

```
#_4.3_########################## VERSIO 3 TARKASTELU ##########################

AreaArrayDict3=[]
for index,element in enumerate(zones):
    element = element.elementId
    Pinta_ala=round((float(Areavalue[index].propertyValues[0].propertyValue.value)),2)
    Tasotieto= str(ZoneLayerValue[index].propertyValues[0].propertyValue.value)
    Vyöhyketieto=str(ZoneNameValue[index].propertyValues[0].propertyValue.value)
    Projektitieto=str(ProjectNameValue[index].propertyValues[0].propertyValue.value)
    Kerrostieto=str(StoryNameValue[index].propertyValues[0].propertyValue.value)
    Versiotieto=str(PhaseValue[index].propertyValues[0].propertyValue.value)
    if ('#Tilataso#'==Tasotieto and ('VE3' in Versiotieto) and '#Kerros#'==Kerrostieto
        and Boksirajat==True):
        ZoneDict3={Vyöhyketieto:Pinta_ala}
        AreaArrayDict3.append(ZoneDict3)
        print("Ominaisuusarvo ZoneDict3:",ZoneDict3,"Ymin:",Boksit_yMin,"Ymax:",Boksit_yMax,
              "Xmin:", Boksit_xMin,"Xmax:",Boksit_xMax)
    else:
        remove
print("------------------------")
print("AreaArrayDict3-listan sisältö:",AreaArrayDict3)
print("------------------------")

for d3 in AreaArrayDict3:
    counter.update(d3)
result3 = dict(counter)
print("Dictionary3-objektin sisältö:", result3)
values3, counts3 = zip(*counter.most_common())
print("------------------------")
print("Esiintyvät tilanimet:",values3)
print("Pinta-alat (saman nimiset yhdistetty):", counts3)
counter.clear()
print("------------------------")
```

Kuva 26. Aliosa 4.3.

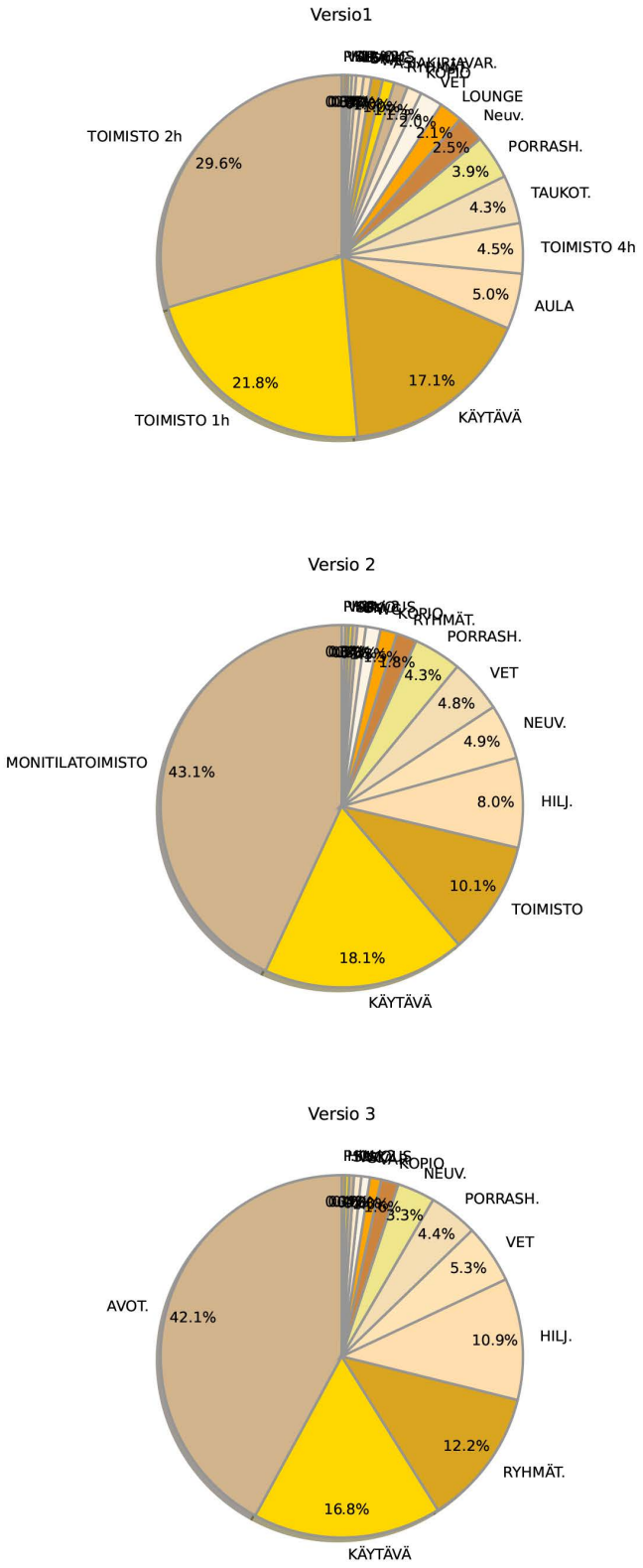
Kuvissa 25 ja 26 olevissa osan 4 aliosissa ohjelma tarkastaa rajaavien muotojen sisällä esiintyvien eri versioiden vyöhyke-elementtien nimet, jotka täyttävät osassa 3 määritellyt ehdot. Ohjelma sitoo samannimiset vyöhykkeet versiokohtaisesti yhteen nimen sekä pinta-alan osalta. Nämä tiivistykset muuttuvat *”key:value”* tyyppisiksi *”dictionary”*-objekteiksi graafista esitystä varten. Laskin nollataan jokaisen vaiheen lopussa *”clear”*-komennolla, jotta vain tarkasteltavan version pinta-alat tulevat huomioitua. Ilman *”clear”*-komentoa, suhteet vääristyisivät versio versiolta. *”float”* ja *”str”* -muunnoksilla osoitetaan se, minkälaisena projektista luettava data käsitellään. *”float”* on lukutyyppi, jossa voidaan esittää desimaaleja. *”str”* merkitsee sisällön muuttamista tavalliseksi merkkijonoksi. Näissä aliosissa minun olisi ollut mahdollista tiivistää ohjelmakoodia, mutta toistolla sisällön hallinta muita käyttäjiä varten oli mielestäni selkeämpi.

```
#_5_########################## KAAVIOIDEN LUONTI ##########################

litterat1 = values1
määrät1 = counts1
litterat2 = values2
määrät2 = counts2
litterat3 = values3
määrät3 = counts3
Otsikko1="Tilojen pinta-alajakauma. YHT:"
Otsikko2="m²"
colors = ['tan','gold','goldenrod', 'navajowhite', 'moccasin', 'wheat','khaki',
          'peru', 'orange','oldlace','blanchedalmond']

fig, axes = plt.subplots(3, 1,figsize=(7,14))
wdges, labels, autopct = axes[0].pie(määrät1,labels=litterat1,radius=1,
                         startangle=90,colors=colors, counterclock=True,
                         shadow=True,wedgeprops={'edgecolor': 'grey',
                         'linewidth': 1},textprops={'fontsize': 6},pctdistance=0.85,
                         autopct='%1.1f%%')
wdges, labels, autopct = axes[1].pie(määrät2,labels=litterat2,radius=1,
                         startangle=90,colors=colors, counterclock=True,
                         shadow=True,wedgeprops={'edgecolor': 'grey',
                         'linewidth': 1},textprops={'fontsize': 6},pctdistance=0.85,
wdges, labels, autopct = axes[2].pie(määrät3,labels=litterat3,radius=1,
                         startangle=90,colors=colors, counterclock=True,
                         shadow=True,wedgeprops={'edgecolor': 'grey',
                         'linewidth': 1},textprops={'fontsize': 6},pctdistance=0.85,
                         autopct='%1.1f%%')

ax = axes[0]
ax.xaxis.set_label_position('top')
ax.set_xlabel('Versio1', fontsize=7)
ax = axes[1]
ax.xaxis.set_label_position('top')
ax.set_xlabel('Versio 2', fontsize=7)
ax = axes[2]
ax.xaxis.set_label_position('top')
ax.set_xlabel('Versio 3', fontsize=7)

plt.title((Projektitieto,Otsikko1,round((Summa1),2),Otsikko2), fontsize=10,x=0.5, y=3.7)
plt.show()

#_6_########################## OHJELMA PÄÄTTYY ##########################
```

Kuva 27. Osat 5-6.

Kuvassa 27 on nähtävillä osa 5, jossa ohjelma rakentaa kolme ympyrädiagrammia perustuen projektista haettuun dataan. Ympyrädiagrammin siivuille määritellään värit, fonttikoot, tekstien sijainnit sekä itse graafien sijainnit. Kirjoitin ohjelman rakentamaan versioiden diagrammit vertikaaliseen järjestykseen. Vaihtamalla numerot kohdassa *”fig, axes = plt.subplots”* *3* ja *1* keskenään, diagrammit tulostuisivat yhdelle riville horisontaalisesti.

('#Projekti Case 2#', 'Tilojen pinta-alajakauma versioittain. 2. Kerroksen pinta-ala:', 906.47, 'm²')



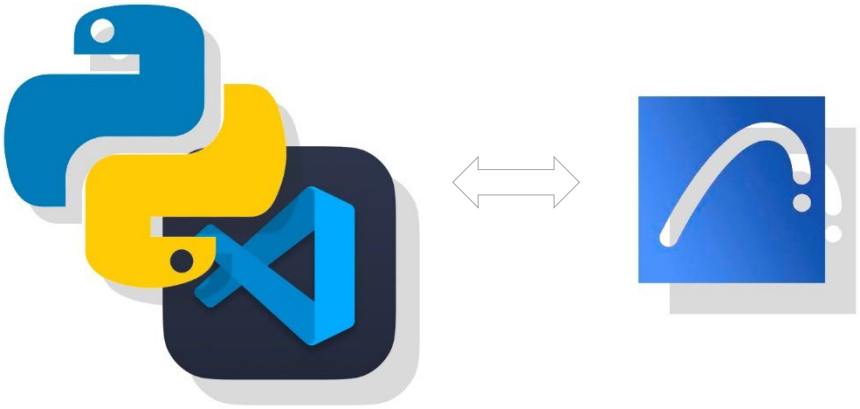
Kuva 28. Diagrammit.

Ohjelman ajon jälkeen ruudulle ilmestyy automaattisesti kuvan 28 (sivulla 63) mukainen graafi, joka on mahdollista tallentaa koneelle. Graafin tiedot ovat nopeasti päivitettävissä ajamalla ohjelma uudelleen. Kuvasta on kuitenkin nähtävissä, että esitystavassa on parannettavaa. Kirjoittaessani koodia, en huomioinut tekstien mahdollista limittymistä päällekkäin. Tämä on kuitenkin korjattavissa kirjoittamalla ohjelmaan koodi, joka esimerkiksi sijoittelee tekstit ennalta määrättyyn sijaintiin tai rakentaa diagrammien viereen värilegendan, jossa tilatyyppien nimet sekä nimiä vastaavat värit esitettäisiin riveittäin.

Kuvan 28 diagrammit olisi toki mahdollista rakentaa muulla tavalla, esimerkiksi käyttäen taulukko-ohjelma Exceliä tai jos nähdään tarve visuaalisesti näyttävämmälle esitystavalle niin tällöin halutussa kuvankäsittelyohjelmassa. Näissä tapauksissa kuitenkin työmäärää lisää erillisen ohjelman käyttö, johon tiedot pitää erikseen päivittää muutoksien sekä tarkennusten yhteydessä. Riippuen käytetystä ohjelmasta päivitys voi olla hidas toimenpide.

Pienillä parannuksilla Python-koodilla rakentuvat diagrammit ovat mielestäni tehokkain ja nopein tapa luoda tarvittaessa case-tapauksen mukaisia esityksiä. Matplotlib-kirjasto mahdollistaa useiden erityyppisten esitystapojen luonnin. Ympyrädiagrammien lisäksi sen avulla voidaan esittää data 2D- tai 3D-muodossa esimerkiksi pylväinä, pisteinä, janana ja niin edelleen. Visuaalisuutta, kuten varjostukset ja värit on mahdollista määritellä koodin kirjoitusvaiheessa, joten graafien jälkikäsittely erillisessä ohjelmassa ei ole välttämätöntä.

4.3 Case 3: VSS-Laskuri



Case-tapauksessa tehtävänäni oli kirjoittaa Pythonilla VSS-laskuri, joka hyödyntäisi paremmin mallista saatavan datan ilman ylimääräistä manuaalista laskentavaihetta. Laskurin käyttämät laskentaperusteet on määritelty RT-kortissa 92 - 11173. Aloitin lähtötietojen keruun selvittämällä toteutuneiden projektien väestösuojien kokoja. Tarvitsin kirjoittamista varten tiedon siitä, kuinka useasti suojatilavaade ylittää S1-luokan väestönsuojan yläraja-arvon. Selvitysten perusteella päädyin toteuttamaan laskuriohjelman soveltuvaksi aluksi vain S1-suojaluokalle.

Kirjoitin ohjelman kuitenkin niin, että laskurin koodi on helposti muutettavissa, jotta suurempien kuin S1-luokan suojien laskeminen olisi mahdollista. Pienet lisäykset ohjelmaan olisivat myös tarpeen, jos ohjelmaan haluttaisiin lisätä ominaisuuksia, jotka tarkastavat muita väestönsuojaan liittyviä tarpeita. Tällaisia lisätarkasteluja voisivat olla esimerkiksi VSS-tilan riittävä korkeus, tarvittavat vesi- ja jätevesiastiat, IV-venttiilien määrät jne. Ohjelma ei ole kuitenkaan pelkkä laskuri vaan kirjoitin siihen lisäksi tarkastusvaiheen itse VSS-vyöhyke-elementille. Jos Archicad-projektissa on edellä mainittu vyöhyke mallinnettuna, ohjelma vertaa laskurin tulosta mallinnetun vyöhykkeen pinta-alaan. Lopuksi ohjelma myös syöttää lasketut tiedot ennalta määrättyihin VSS-vyöhyke-elementin ominaisuuskenttiin.

Ohjelman kirjoittamiseen sekä testiajoihin aikaa kului noin kolme tuntia. Testiajojen yhteydessä eteeni tuli ongelma, jossa ohjelma ajautui loputtomaan toistosilmukkaan, kun yritin ajaa sitä Archicadin oman Python-konsolin kautta. Tämä johtui kirjoittamastani kohdasta, jossa laskuria käyttävän henkilön tulee kirjoittaa RT-kortissa määritelty prosenttimäärää, jonka mukaan VSS-tilatarpeen laskenta suoritetaan. Tämän testiajon ansiosta minulle paljastui, ettei Archicadin oma Python-konsoli tue käyttäjäsyötettä sen jälkeen, kun ohjelman ajo on kerran aloitettu.

Ratkaisin ongelman kirjoittamalla ohjelman ajoon erillisen .bat-tiedoston. Se sisältää komennon, jolla ohjelma aukeaa erilliselle komentoriville, jossa käyttäjän on mahdollista kirjoittaa tarvittava merkkisyöte. Näin IDEä ei välttämättä tarvita ohjelman ajamiseen ja Archicadin Python-konsolista johtuva ongelma voitiin kiertää.

```
#_1_############################ KYTKENNÄT ############################
from os import remove
from archicad import ACConnection
import archicad
import sys
import re
conn = ACConnection.connect()

from archicad.releases.ac25.b3000types import ElementId, EnumValueId
conn = ACConnection.connect()

assert conn

acc = conn.commands
act = conn.types
acu = conn.utilities

#_2_############################ ALKUMÄÄRITYKSET ############################

zones=acc.GetElementsByType('Zone')
Type=acu.GetBuiltInPropertyId('General_Type')
Area=acu.GetBuiltInPropertyId('Zone_MeasuredArea')
Zonetype=acu.GetBuiltInPropertyId('Zone_ZoneName')
StoryName=acu.GetUserDefinedPropertyId('#Ominaisuusryhnmä1#','Sijoituskerros')
VssVaade=acu.GetUserDefinedPropertyId('#Ominaisuusryhnmä2#','VSS Vaade')
Vsskäymälät=acu.GetUserDefinedPropertyId('#Ominaisuusryhnmä2#','Käymälät')
VssIVKoneet=acu.GetUserDefinedPropertyId('#Ominaisuusryhnmä2#','IV-koneet')
VssSulkutila=acu.GetUserDefinedPropertyId('#Ominaisuusryhnmä2#','Sulkutila')
Areavalue=acc.GetPropertyValuesOfElements(zones,[Area])
ZoneTypeValue=acc.GetPropertyValuesOfElements(zones,[Zonetype])
StoryNameValue=acc.GetPropertyValuesOfElements(zones,[StoryName])
```

Kuva 29. Osat 1–2.

Kuvassa 29 on nähtävillä ohjelman osat 1 ja 2. Näiden osien toimintaperiaate on käytännöllisesti katsoen sama kuin aikaisempien case-tapausten kohdalla. Muuttujien sisältö on kuitenkin jälleen erilainen johtuen ohjelman käyttötarkoituksesta.

```
#_3_################### PINTA-ALOJEN TARKISTUKSET SEKÄ YHTEENLASKUT ########################


print("----------------------------")

AreaSum =[]
for index,element in enumerate(zones):
    element = element.elementId
    PintaAlat3= float(Areavalue[index].propertyValues[0].propertyValue.value)
    Vyöhyketyyppi= str(ZoneTypeValue[index].propertyValues[0].propertyValue.value)
    Kerrostieto=str(StoryNameValue[index].propertyValues[0].propertyValue.value)
    if "#Vyöhykenimi1#"==Vyöhyketyyppi:
        AreaSum.append(PintaAlat3)
        print("Sijoituskerros:",Kerrostieto,", Vyöhyketyyppi:",Vyöhyketyyppi,", Ala:",
              PintaAlat3,"m²")
    else:
        remove
Summa=sum(AreaSum)
print("Kerrosalojen summa on:", Summa)

VSSArea=[]
for index,element in enumerate(zones):
    element = element.elementId
    PintaAlat3= float(Areavalue[index].propertyValues[0].propertyValue.value)
    Vyöhyketyyppi= str(ZoneTypeValue[index].propertyValues[0].propertyValue.value)
    if "#Vyöhykenimi2#"==Vyöhyketyyppi:
        VSSArea.append(PintaAlat3)
    else:
        remove
Summa2=sum(VSSArea)

print("----------------------------")


#_4_########################## KÄYTTÄJÄN SYÖTTÄMÄ %-ARVO ##########################

print("Varsinaisen suojatilan pinta-ala kerrosalasta prosentteina
     (2 pros. rakennuksen kerrosala. 1 pros. myymälä-, teollisuus-,
     tuotanto-, varasto-, kokoontumisrakennus):")
while True:
    try:
        pros=int(input())
    except ValueError:
        print("Väärä merkki. Tarkista syöttämäsi arvo.")
        continue
    if (pros!=1 and pros!=2):
        print("Väärä prosenttimäärä. Tarkista syöttämäsi arvo.")
        continue
    else:
        break

print("----------------------------")
```

Kuva 30. Osat 3–4.

Kuvassa 30 ovat ohjelman osat 3 ja 4. Osassa 3 ohjelma tarkastaa projektista löytyvien kerrosalavyöhykkeiden sijaintikerrokset sekä laskee niiden pinta-alojen summan muistiin muuttujaan *”Summa”*. Samassa osassa ohjelma tarkastaa myös, onko projektissa mallinnettua VSS-

vyöhykettä. Jos kyseinen vyöhyke löytyy, sen pinta-ala sijoitetaan muistiin muuttujaan ”Summa2”.

Osassa 4 ohjelma tarkastaa käyttäjän syöttämän prosenttiluvun kokonaislukuna, jonka mukaan seuraavassa osassa oleva suojahuoneen pinta-alavaateen lasku suoritetaan. Ohjelma antaa virheilmoituksen, jos käyttäjä syöttää minkä tahansa muun merkin tai luvun kuin *”1”* tai *”2”*. Ohjelma toistaa tämän pyynnön niin kauan kunnes oikea luku syötetään. Tästä osasta johtui case-tapauksen alussa kertomani loputon toistosilmukkaongelma, jos ajo tehdään Archicadin omassa Python-konsolissa.

Toistosilmukkaongelma viimeistään osoitti minulle, että vaikka Archicadin oma Python-konsoli on hyvä lisä, en voi täysin suositella sen käyttämistä. Case-projektien testiajoja tehdessäni huomasin myös, että kirjoittamani ohjelmat suorittivat kirjoitetun tehtävän hitaammin Archicadin omasta Python-konsolista kuin ajaessani ohjelmaa oikean IDEn kautta tai .bat-tiedostosta.

Samantapaisia ongelmia voi ilmetä myös niissä suunnittelusovelluksissa, joissa on käytettävissä sovelluksen sisäinen Python-konsoli. Tämän takia suosittelen, että kirjoitettaessa ja ajettaessa ohjelmia, siihen käytetään tarkoituksenmukaista IDEä ja tarvittaessa .bat-tiedostoja.

```
#_5_########################## VSS-LASKELMAT JA TARKASTELUT ##########################

#_5.1_########################## SUOJATILA ##########################

Suojatila=round(Summa*(int(pros)/100))
HenkilöMäärä=round(Suojatila/0.75)
if Suojatila<=90:
    print("Varsinainen suojatila:", Suojatila,"m². Henkilömäärä", HenkilöMäärä)
elif (Suojatila>90 and round(Suojatila-90)<=45):
    print("Tarvitaan 2 suojahuonetta. Suojatilan koko", Suojatila,"m². Henkilömäärä",
HenkilöMäärä)
elif (Suojatila>135 and round(Suojatila-90)>45):
    print("Tarvitaan vähintään 2 kpl S1-luokan suojia, S2-luokan suoja (max 900 m²)
          tai kalliosuoja (max 4500 m²). Suojatilan koko:", round(Suojatila),"m².
          Ohjelma päättyy.")
    quit
print("-----------------------------")

#_5.2_########################## IV-KONEET ##########################
if Suojatila<=135:
              if Suojatila<45:
                  ivk=1
              elif 90>Suojatila>=45:
                  ivk =2
              elif 135>Suojatila>=90:
                  ivk =3
              elif Suojatila==135:
                  ivk =4
              print("Ilmanvaihtokoneita tarvitaan",ivk,"kpl.")

#_5.3_########################## KÄYMÄLÄT ##########################
if Suojatila<=135:
              if Suojatila<40:
                  wc=2
              elif 52>Suojatila>=40:
                  wc=3
              elif 72>Suojatila>=52:
                  wc=4
              elif 100>Suojatila>=72:
                  wc=5
              elif 120>Suojatila>=100:
                  wc=6
              elif 140>Suojatila>=120:
                  wc=7
              print("Kuivakäymälöitä tarvitaan",wc,"kpl.")

#_5.4_########################## SULKUTILA ##########################

if Suojatila<90:
    Sulkutila=2.5
    print("Sulkuteltta. Tilatarve vähintään", Sulkutila, "m²")
    print("-----------------------------")
if 135>=Suojatila>=90:
    Sulkutila=2.5
    print("Sulkuteltta tai -huone. Tilatarve vähintään", Sulkutila, "m²")

    print("-----------------------------")
```

Kuva 31. Aliosat 5.1–5.4.

Kuvassa 31 on ohjelman osa 5 ja kyseisen osan aliosat. Aliosissa lasketaan projektista haetun kerrosaladatan perusteella suojatilan vaade, henkilömäärä, sulkutila sekä tarvittavat IV-koneet ja

kuivakäymälät. Laskin pysähtyy ja ohjelma päättyy aliosaan 5.1 jos suojatilavaade ylittää arvon 135 m². Laskureissa olevat raja-arvot sekä määrät on otettu RT-kortin 92 - 11173 ”S1-luokan teräsbetoniväestönsuoja” taulukossa 2 esitetyistä arvoista ja luvuista.

```
#_6_############################ LASKELMIEN YHTEENVETO, PROJEKTIN VSS-VYÖHYKKEEN TARKASTUS SEKÄ
TIETOJEN AJO VYÖHYKKESEEN SEN LÖYTYESSÄ PROJEKTISTA ############################

if Suojatila<135:
    print("Pinta-alalaskelmat:\n\nVarsinainen suojatila:",Suojatila,"m²\n
          IV-koneet:",round(1.5* ivk,1),"m²\nSulkutila:",Sulkutila,"m²\n")
    Laskuri=(Suojatila+round(1.5*ivk,1)+Sulkutila)
    print("-----------------------------")
    print("Väestönsuojan tilavaraus vähintään:", Laskuri, "m²")
    print("-----------------------------")
    if Summa2>0:
        print("Projektissa on #Ominaisuusryhnmä2#, jonka koko on",round(Summa2),"m²")
        if Summa2>=Laskuri:
            print("Tilavaraus on riittävä.")
            print("-----------------------------")
            VSSArray = []
            for index,element in enumerate(zones):
                element = element.elementId
                Vyöhyketyyppi= str(ZoneTypeValue[index].propertyValues[0].propertyValue.value)
                SuojatilaSTR=((str(Suojatila))+" m²")
                IVkoneetSTR=((str(ivk))+" KPL, ALA: "+str(round(1.5*ivk,1))+" m²")
                KäymälätSTR=((str(wc))+" KPL, ALA: "+str(round(0.7*wc,1))+" m²")
                SulkuSTR=(str(Sulkutila)+ " m²")
                if "#Vyöhykenimi2#"==Vyöhyketyyppi:
                    VSSPintaAla = act.NormalStringPropertyValue(SuojatilaSTR,type='string',
                                  status = 'normal')
                    VSSIVkone = act.NormalStringPropertyValue(IVkoneetSTR,type='string',
                                status = 'normal')
                    VSSkäymälä = act.NormalStringPropertyValue(KäymälätSTR,type='string',
                                 status = 'normal')
                    VSSsulku = act.NormalStringPropertyValue(SulkuSTR,type='string',
                               status = 'normal')
                    VSSarvo1 =act.ElementPropertyValue(element, VssVaade, VSSPintaAla)
                    VSSarvo2 =act.ElementPropertyValue(element, VssIVKoneet, VSSIVkone)
                    VSSarvo3 =act.ElementPropertyValue(element, Vsskäymälät, VSSkäymälä)
                    VSSarvo4 =act.ElementPropertyValue(element, VssSulkutila, VSSsulku)
                    VSSArray.append(VSSarvo1)
                    VSSArray.append(VSSarvo2)
                    VSSArray.append(VSSarvo3)
                    VSSArray.append(VSSarvo4)
                    tulos2 = acc.SetPropertyValuesOfElements(VSSArray)
                else:
                    remove
        else:
            print("RIITTÄMÄTÖN TILAVARAUS! VSS-tilaa on kasvatettava vähintään",
                  round(Laskuri-Summa2),"m² verran.")
            print("-----------------------------")
    elif Summa2==0:
        print("PROJEKTISTA EI LÖYDY VSS-VYÖHYKETTÄ TARKASTUSTA VARTEN!")
        print("-----------------------------")
#_7_############################ OHJELMA PÄÄTTYY ############################
```

Kuva 32. Osat 6–7.

Kuvassa 32 sivulla 71 on esitetty ohjelman osat 6 ja 7. Osassa 6 ohjelma tulostaa laskennan tulokset konsoliin riveittäin. Tässä osassa ohjelma

tarkastaa myös mallissa olevan VSS-vyöhykkeen pinta-alan ja vertaa sitä osassa 5 laskettuihin arvoihin. Jos projektissa oleva VSS-vyöhyke on kooltaan riittämätön tai se puuttuu kokonaan, tulostuu konsoliin tästä ilmoitus. Vyöhykkeen jalanjäljen ollessa riittävä, tulostuu kuittaus konsoliin. Osaan 6 voidaan tarvittaessa esimerkiksi lisätä myös tarkistus tilan korkeudesta käyttämällä *”Get3DBoundingBoxes”* komentoa. Osa 7 on merkintä ohjelman päättymisestä kyseiseen kohtaan.

```
----------------------------
Sijoituskerros: 5. Kerros , Vyöhyketyyppi: Kerrosala, Ala: 611.438392039 m²
Sijoituskerros: 1. Kerros , Vyöhyketyyppi: Kerrosala, Ala: 865.885827897 m²
Sijoituskerros: 4. Kerros , Vyöhyketyyppi: Kerrosala, Ala: 984.304165443 m²
Sijoituskerros: 3. Kerros , Vyöhyketyyppi: Kerrosala, Ala: 984.342488714 m²
Sijoituskerros: 2. Kerros , Vyöhyketyyppi: Kerrosala, Ala: 984.10487736 m²
Kerrosalojen summa on: 4430.075751453
----------------------------
Varsinaisen suojatilan pinta-ala kerrosalasta prosentteina (2 pros. rakennuksen
kerrosala. 1 pros. myymälä-, teollisuus-, tuotanto-, varasto-, kokoontumisrakennus):
2
----------------------------
Varsinainen suojatila: 89 m². Henkilömäärä 119
----------------------------
Ilmanvaihtokoneita tarvitaan 2 kpl.
Kuivakäymälöitä tarvitaan 5 kpl.
Sulkuteltta. Tilatarve vähintään 2.5 m²
----------------------------
Pinta-alalaskelmat:

Varsinainen suojatila: 89 m²
IV-koneet: 3.0 m²
Sulkutila: 2.5 m²

----------------------------
Väestönsuojan tilavaraus vähintään: 94.5 m²
----------------------------
Projektissa on VSS-vyöhyke, jonka koko on 97 m²
Tilavaraus on riittävä.
----------------------------
```

Kuva 33. VSS-laskurin ajo komentokehotteessa.

Kuvassa 33 on esitetty .bat-tiedoston kautta käynnistetty komentokehotteessa toimiva laskuriohjelman ajotila. Jos en olisi testiajojen aikana törmännyt toistosilmukkaongelmaan, en olisi edes tutkinut tätä vaihtoehtoa, jossa ohjelman ajoon käytetään erillistä käynnistystiedostoa. .bat-tiedostojen vahvuus on siinä, että ne ajavat ohjelman todella nopeasti, ovat keveitä ja eivät vaadi erillistä

asennusta toimiakseen. Jos ohjelmakoodia ei tarvitse tarkastella tai muokata, niin tällöin on parempi käyttää .bat-tiedostoa ajoa varten IDEn sijaan. Jälkeenpäin tarkasteltuna voinkin todeta, että ongelman ilmenemisestä oli huomattavasti enemmän hyötyä kuin haittaa. Käyttämäni ratkaisu on hyödynnettävissä kaikentyyppisten .py-tiedostojen kohdalla. Myös sellaisten, joissa ohjelma ei vaadi ajon aikana käyttäjän kirjoittamaa syötettä.

Kuva 34. Vyöhyke-elementin ominaisuustiedot

Kun ohjelma on saavuttanut päätöspisteensä onnistuneesti, tulosdata siirtyy Archicad-projektissa ennalta määrättyihin väestönsuojavyöhykkeen ominaisuuskenttiin (Kuva 34). Nämä tiedot voidaan taulukoida Archicadissa, sijoittaa tarpeellisiin piirustuksiin sekä saada kulkemaan myös IFC-tallenteen mukana.

VSS-laskuri oli case-tapauksiin kirjoittamistani ohjelmista helpoin toteuttaa. Sen toiminta ei ollut riippuvainen erillisistä Excel-taulukoista tai lisäosista. Siksi ohjelman rakenne oli minusta mielenkiintoinen, koska rivipituutta koodille kuitenkin kertyi yllättävän paljon verrattuna muiden case-tapausten ohjelmiin. Toistosilmukkaongelma oli pienimuotoinen harmi, mutta minulle iso osoitus siitä kuinka tärkeää on suorittaa testiajot mahdollisimman monella tavalla ennen ohjelman jakamista yleiseen käyttöön.

5 JOHTOPÄÄTÖKSET

Suunnittelutyössä painottuu päivä päivältä enemmän tietomallien merkitys. Niiden tarkkuustasot sekä vaatimukset ovat suoraan kytköksissä projektin kokoluokkaan. Käytännössä voidaan todeta, että mitä suurempi projekti, sen tarkempi itse mallin tulee olla. Tarkkuustason kasvulla on kuitenkin suora vaikutus aikatauluihin, jos tietomallin vaadittu tietosisältö kirjoitetaan manuaalisesti mallin elementteihin. Koska projektiaikataulut ovat tänä päivänä kireitä, automaatio voi olla yksi ratkaisu suorittamaan ne tehtävät, jotka voidaan järkevästi itse ohjelmoimalla automatisoida.

Diplomityössäni tutkin onko ohjelmointitaito tarpeellinen työväline arkkitehdin ammatissa. Lähestyin tutkittavaa aihetta tietomallintamisen näkökulmasta. Tutkimusvaiheessa selvitin millä tavalla ohjelmointia ja sen opetusta on aikaisemmin lähestytty arkkitehtikoulutuksessa sekä arkkitehdin varsinaisissa työtehtävissä, minkälaisissa tilanteissa koodausta on pääasiassa hyödynnetty sekä missä yhteydessä ohjelmointia olisi mahdollista tällä hetkellä käyttää. Hain vastauksia kysymyksiin, onko ohjelmoinnin opiskelu arkkitehdeille hyödyllistä sekä mitä ohjelmoinnin avulla voisi arjen tehtävissä saavuttaa. Työvaiheessa ajatukset ohjelmoinnista konkretisoituivat itse kirjoitettujen ohjelmien muodossa. Niiden kirjoittamista sekä käyttämistä suunnittelutyön osana tarkastelin case-tapausten kautta.

Ohjelmointia arkkitehdit ovat aikaisemmin hyödyntäneet työvälineenä isommassa mittakaavassa lähinnä VPL-kieliä tukevien sovellusten yhteydessä. Tällaisia sovelluksia ovat esimerkiksi Grasshopper sekä Dynamo. VPLn hyödyntäminen on ollut kuitenkin järkevää vain tilanteissa, joissa muodon tuottaminen perinteisemmillä mallinnustekniikoilla on ollut haasteellista ja aikaa vievää. Mahdollisesti tämän vuoksi 3. luvussa mainitsemilleni VPL-kursseille osallistuneet oppilaat kokivat, etteivät tule käyttämään ohjelmointia työtehtävissään, vaikka se herättikin heissä mielenkiintoa. VPL-kieliä tukevien sovellusten avulla ei ole tällä hetkellä päivittäisessä suunnittelutyössä useinkaan mahdollista tuottaa käyttökohteiden kannalta merkittävää hyötyä.

Vielä muutama vuosi sitten ohjelmoinnin tarpeellisuudesta arkkitehtisuunnittelussa olisi siis voinut todeta, että ohjelmointiosaamista ei tarvita. Päivittäiseen työskentelyyn sitä ei ollut mahdollista sisällyttää järkevällä tavalla. Sovellukset kuitenkin kehittyvät ja monipuolistuvat nykypäivänä nopeasti. Suunnittelu-sovellukset ovat viime aikoina avanneet rajapintojaan varsinkin Python-kielelle. Automatisoinnin kannalta rajapintojen avaaminen on ollut merkittävä kehityssuunta. Juuri siksi ohjelmointiosaamisen tarpeellisuuden arviointi tässä hetkessä on tärkeää. Tietomallintamisen näkökulmasta ohjelmointitaito antaa merkittävän edun ja hyödyn arkkitehdille. Tehtävät, jotka toistuvat projekteista toiseen samanlaisena voitaisiin antaa itse kirjoitetun ohjelman hoidettavaksi. Tällöin edellä mainituista tehtävävaiheista säästyvä aika pystyttäisiin kohdentamaan varsinaisten suunnitteluongelmien ratkaisemiseen.

Hyöty ei ole kuitenkaan saavutettavissa ilman ajallista panostusta. Alussa aikaa kuluu paljon käytettävän ohjelmointikielen opiskeluun ennen kuin ensimmäistäkään omaa työtä hyödyntävää sovellusta voidaan kirjoittaa. Toisaalta osaamisen kasvun voi kuitenkin sanoa olevan eksponentiaalista. Mitä enemmän kirjoitat ohjelmia, sitä pienemmässä aikaikkunassa osaat kirjoittaa niistä laajempia, selkeämpiä sekä opit helpommin hyödyntämään sisältöä, joka tulee suunnittelusovelluksen ulkopuolelta. Ohjelmoinnin sekä automatisoinnin hyödyntäminen onnistuu myös sitä paremmin, mitä tarkemmin malli on rakennettu. Jos suunnitelmat koostuvat pääsääntöisesti 2D-viivoista, ohjelmoinnin avulla ei saavuteta työskentelyn kannalta juuri minkäänlaista lisäarvoa tai ajallista hyötyä.

Kun ajallinen panostus kielen opiskeluun on käytetty, tulisi pitää huolta, ettei sen avulla säästetty aika valu tulevaisuudessa hukkaan kiristyneiden projektiaikataulujen muodossa. Tämä on suurin yksittäinen riskitekijä, kun mietitään ohjelmointitaidon mukanaan tuomia mahdollisia haittoja. Kiristynyt aikataulu voi pahimmillaan johtaa noidankehään, jossa projektiin on pakko kirjoittaa uusia ohjelmia, jotta se olisi edes teoriassa mahdollista saada päätökseen annetussa ajassa, mutta tarvittavien ohjelmien koodaamiseen ei ole varsinaisesti aikaa käytettäväksi. Tällöin hyödyllinen taito onkin vain raskaalta sekä

turhauttavalta tuntuva ylimääräinen työvaihe. Varsinkin silloin, jos kirjoitettua ohjelmaa ei saada toimimaan halutulla tavalla.

Toinen riskitekijä liittyy itse ohjelmien käytettävyyteen. Jos tavoitellaan tilannetta, että mahdollisimman moni pystyisi itse muokkaamaan sekä ymmärtämään aikaisemmin kirjoitettujen ohjelmien sisältöä, koodien ei tulisi olla liian virtaviivaistettuja. Koodatun sisällön pitäisi olla siis sillä tasolla, että peruskäyttäjän ei tarvitsisi panostaa suuria määriä aikaa kielen opiskeluun. Kirjoitettujen ohjelmien tulisi myös olla tekijän toimesta riittävän hyvin dokumentoituja, jotta ongelmatilanteissa sisällön tulkitsemiseen ei tarvittaisi välttämättä ohjelman koodanneen henkilön apua.

Pienempiä riskitekijöitä voivat olla esimerkiksi muutosvastarinta tai yrityksen käyttämä IT-infrastruktuuri. Jos työympäristö ei koe kirjoitettua koodia ja automatisaatiota omaa työskentelyä helpottavana tekijänä vaan ennemminkin epäilyttävänä tuntemattomana, ohjelmoinnin hyötyjä voi olla hyvin vaikea jalkauttaa päivittäiseen käyttöön. IT:n osalta ongelmia voivat tuoda erilaiset estot. Näitä estoja voivat olla esimerkiksi estetyt tai kielletyt ohjelmat yrityksen verkossa, ohjelmien asennusoikeuksien rajoittaminen ja niin edelleen.

Miettiessäni vastausta kysymykseen tulisiko arkkitehdin osata ohjelmoida, jouduin puntaroimaan edellisissä luvuissa käsiteltyjä asioita sekä tarkastelemaan case-tapauksiin kirjoittamiani ohjelmia. Riskeistä huolimatta päädyin pohdinnoissani myöntävään vastaukseen. Suurimpana vaikuttajana päätökseen oli case-tapauksissa käytetty ohjelmointikieli sekä sen avulla mahdollistettava automatisointi. Python on hyvin dokumentoitu, helposti omaksuttava ja sen käytettävyys ei rajaudu pelkästään Archicadiin, vaan sitä voidaan hyödyntää esimerkiksi Revitissä, Dynamossa tai Grasshopperissa. Toisena vaikuttajana oli jatkokäyttö. Kun projektit rakennetaan samalta pohjalta, kerran kirjoitettua ohjelmaa voidaan ajaa uudestaan ilman muutoksia.

Python täyttää siis ne kaksi vaatimusta, jotka Streich artikkelissaan vuodelta 1992 asetti ohjelmoinnin käytölle arkkitehtisuunnittelussa. Ensimmäinen vaatimus oli se, että käytetyn kielen tulee olla tarpeeksi yksinkertainen sekä monikäyttöinen. Toinen vaatimus oli, että

suunnittelijan tulee itse pystyä kirjoittamaan ohjelmoitavan suunnittelutehtävän vaatima koodi. Jos tulevaisuudessa suunnittelusovellusten rajapintoja avataan vieläkin enemmän itse koodatuille ohjelmille, ohjelmointiosaamisesta tulee entistä tärkeämpi sekä hyödyllisempi työväline arkkitehdin arjessa.

LÄHTEET

Aatsalo, J. (18.12.2020). Tutkimus: Rakennusten digitaaliset kaksoset edistyvät hitaasti – käyttäjät haluavat yksinkertaisuutta. *Rakennuslehti.* https://www.rakennuslehti.fi/2020/12/tutkimus-rakennusten-digitaaliset-kaksoset-edistyvat-hitaasti-kayttajat-haluavat-yksinkertaisuutta/

Agirbas, A. (2017). Teaching Design by Coding in Architecture Undergraduate Education. *Future Trajectories of Computation in Design [17th International Conference, CAAD Futures 2017, Proceedings / ISBN 978-975-561-482-3] Istanbul, Turkey, July 12-14, 2017*. 249-258. http://papers.cumincad.org/cgi-bin/works/paper/cf2017_249

Aguiar, R., Gonçalves, A. (2015). Programming for Architecture: The Students' Point of View. *Real Time - Proceedings of the 33rd eCAADe Conference - Volume 2.*159-168. http://papers.cumincad.org/cgi-bin/works/paper/ecaade2015_278

Boudreau, E. (16.10.2020). What is A Programming Paradigm? *towards data science. https://towardsdatascience.com/what-is-a-programming-paradigm-1259362673c2*

Burry, M. (1997). Narrowing the Gap Between CAAD and Computer Programming: A Re-Examination of the Relationship Between Architects as Computer-Based Designers and Software Engineers, Authors of the CAAD Environment. *CAADRIA '97,* 491–498. http://papers.cumincad.org/cgi-bin/works/Show?4b42

Celani, G., Vaz C. E. V. (2012). CAD Scripting and Visual Programming Languages for Implementing Computational Design Concepts: A Comparison from a Pedagogical Point of View. *International Journal of Architectural Computing vol. 10 - no. 1,* 121-138. http://papers.cumincad.org/cgi-bin/works/paper/ijac201210108

Celani, M. G. C. (2008). Teaching CAD programming to architecture students. *Gestão & Tecnologia De Projetos (Design Management and Technology), 3(2),* 1-23. https://doi.org/10.4237/gtp.v3i2.73

Cloudpermit. (22.6.2021). *Lupapiste käsitteli maailman ensimmäisen 3D BIM-tietomallinnuksella tehdyn rakennusluvan Järvenpään kaupungille.* https://cloudpermit.com/ajankohtaista/lupapiste-kasitteli-maailman-ensimmaisen-3d-bim-tietomallinnuksella-tehdyn-rakennusluvan-jarvenpaan-kaupungille

Graphisoft. (12.1.2022). *Element ID Conflict Checker.* https://graphisoft.com/downloads/python

Hancock, J. R. (25.12.2014). Margaret Hamilton, the Engineer Who took the Apollo to the Moon. *Verne.*https://verne.medium.com/margaret-hamilton-the-engineer-who-took-the-apollo-to-the-moon-7d550c73d3fa

Jäväjä, P. & Lehtoviita T. (2016). *Tietomallintaminen talonrakennustyömaalla.* Rakennustieto

IBM. (23.12.2021). *FORTRAN The Pioneering Programmin Language.* https://www.ibm.com/ibm/history/ibm100/us/en/icons/fortran/

Klein, B. (17.7.2021). History and Philosophy of Python. *python-course.eu.* https://python-course.eu/python-tutorial/history-and-philosophy-of-python.php

Long, J., Murphy, J., Carnegie, D., Kapur, A. (2017). Loudspeakers Optional: A history of non-loudspeaker-based electroacoustic music. *Organised Sound , Volume 22 , Special Issue 2: Alternative Histories of Electroacoustic Music , August 2017,* 195–205. https://doi.org/10.1017/S1355771817000103

Nokia. (31.1.2006) *Nokia to Release Python for S60 Source Code to Open-Source Software Developer Community.* https://web.archive.org/web/20070518052247/http://press.nokia.com/PR/200601/1032017_5.html

Pyninsula Official. (21.8.2019). *Gui Talarico – Python in Architecture – Pyninsula #20*[video]. YouTube. https://www.youtube.com/watch?v=H6BKT4rLncM

Python. (23.12.2021). *Extending Python with C or C++.* https://docs.python.org/3/extending/extending.html#reference-counts

Streich, B. (1992). Should We Integrate Programming Knowledge into the Architect's CAAD-Education. *CAAD Instruction: The New Teaching of an Architect? [eCAADe Conference Proceedings] Barcelona (Spain) 12-14 November 1992,* 399-409. http://papers.cumincad.org/cgi-bin/works/paper/61e0

Tan, C. (2020). The Poetics of Computer Code: Tracing Digital Inscription in Ada Lovelace's England. *Digital Studies/Le champ numérique, 10(1),* 2–5. https://www.semanticscholar.org/paper/The-Poetics-of-Computer-Code%3A-Tracing-Digital-in-Tan/a85bb99a345946bcf5358eb21550b52841ac2ff8

KUVALÄHTEET

Kuvat 1, 2, 3:

Celani, G., Vaz C. E. V. (2012). CAD Scripting and Visual Programming Languages for Implementing Computational Design Concepts: A Comparison from a Pedagogical Point of View. *International Journal of Architectural Computing vol. 10 - no. 1,* 121-138. Table 3., Figure 8., Figure 9. http://papers.cumincad.org/cgi-bin/works/paper/ijac201210108

Kuva 4:

Aguiar, R., Gonçalves, A. (2015). Programming for Architecture: The Students' Point of View. *Real Time - Proceedings of the 33rd eCAADe Conference - Volume 2.*159-168. Figure 4, Figure 5, Figure 6. http://papers.cumincad.org/cgi-bin/works/paper/ecaade2015_278

Kuva 5:

Agirbas, A. (2017). Teaching Design by Coding in ArchitectureUndergraduate Education. *Future Trajectories of Computation in Design [17th International Conference, CAAD Futures 2017, Proceedings / ISBN 978-975-561-482-3] Istanbul, Turkey, July 12-14, 2017*. 249-258. Fig. 4. http://papers.cumincad.org/cgi-bin/works/paper/cf2017_249

Kuva 6:

Celani, M. G. C. (2008). Teaching CAD programming to architecture students. *Gestão&Tecnologia De Projetos (Design Management and Technology), 3(2),* 1-23. Figure 5. https://doi.org/10.4237/gtp.v3i2.73

```
elem_classes = []
for index,element2 in enumerate(AllElements):
    Tasotieto=str(value[index].propertyValues[0].propertyValue.value)
    Elementtityyppi=str(value2[index].propertyValues[0].propertyValue.value)
    ObjektiNimi=str(value3[index].propertyValues[0].propertyValue)

    ###_4.0_###_YLEISET_####
    if "\x14"==Tasotieto:
        class2 = acu.FindClassificationItemInSystem('#Luokitusjärjestelmä#','#Luokka#')
        class_id = act.ClassificationId(class1,class2.classificationItemId)
        elem_class = act.ElementClassification(element2.elementId,class_id)
        print("#Luokka#",Elementtityyppi,Tasotieto)
    if (
            "\x14"!=Tasotieto and
            "Ikkuna"!=Elementtityyppi and
            "Ovi"!=Elementtityyppi and
            "Aukko"!=Elementtityyppi and
            "Kattoikkuna"!=Elementtityyppi
        ):
        values_of_tasotieto=(str([a_dict[Tasotieto] for a_dict in my_list2])[2:-2])
        luokka=str(values_of_tasotieto)
        class2 =
acu.FindClassificationItemInSystem('#Luokitusjärjestelmä#',values_of_tasotieto)
        class_id = act.ClassificationId(class1,class2.classificationItemId)
        elem_class = act.ElementClassification(element2.elementId,class_id)
        print(luokka,Elementtityyppi,Tasotieto)

    ###_4.1_###_Ikkunat_####
    if "Ikkuna"==Elementtityyppi:
        if ("#Taso#"==Tasotieto and "Ikkuna PK21" in ObjektiNimi):
            class2 = acu.FindClassificationItemInSystem('#Luokitusjärjestelmä#','#Luokka#')
            class_id = act.ClassificationId(class1,class2.classificationItemId)
            elem_class = act.ElementClassification(element2.elementId,class_id)
            print("#Luokka#", Elementtityyppi,Tasotieto)
        elif ("#Taso2"==Tasotieto and "Ikkuna PK21" in ObjektiNimi):
            class2 = acu.FindClassificationItemInSystem('#Luokitusjärjestelmä#','#Luokka#')
            class_id = act.ClassificationId(class1,class2.classificationItemId)
            elem_class = act.ElementClassification(element2.elementId,class_id)
            print("#Luokka#", Elementtityyppi,Tasotieto)
        elif ("#Taso#"==Tasotieto and "Pinta-alue PK21" in ObjektiNimi):
            ObjektiNimi=str(value3[index].propertyValues[0].propertyValue.value)
            class2 = acu.FindClassificationItemInSystem('#Luokitusjärjestelmä#','#Luokka#')
            class_id = act.ClassificationId(class1,class2.classificationItemId)
            elem_class = act.ElementClassification(element2.elementId,class_id)
            print("#Luokka#", Elementtityyppi,ObjektiNimi,Tasotieto)

    ###_4.2_###_OVET_####
    if "Ovi"==Elementtityyppi:
        if ("#Taso#"==Tasotieto):
            class2 = acu.FindClassificationItemInSystem('#Luokitusjärjestelmä#','#Luokka#')
            class_id = act.ClassificationId(class1,class2.classificationItemId)
            elem_class = act.ElementClassification(element2.elementId,class_id)
            print("#Luokka#", Elementtityyppi,Tasotieto)
        elif ("#Taso2"==Tasotieto):
            class2 = acu.FindClassificationItemInSystem('#Luokitusjärjestelmä#','#Luokka#')
            class_id = act.ClassificationId(class1,class2.classificationItemId)
            elem_class = act.ElementClassification(element2.elementId,class_id)
            print("#Luokka#", Elementtityyppi,Tasotieto)

    ###_4.3_###_AUKOT_####
    if "Aukko"==Elementtityyppi:
        if ("#Taso3#" in Tasotieto):
            class2 = acu.FindClassificationItemInSystem('#Luokitusjärjestelmä#','#Luokka#')
            class_id = act.ClassificationId(class1,class2.classificationItemId)
            elem_class = act.ElementClassification(element2.elementId,class_id)
            print("#Luokka#", Elementtityyppi,Tasotieto)
        elif ("#Taso4#" in Tasotieto):
            class2 = acu.FindClassificationItemInSystem('#Luokitusjärjestelmä#','#Luokka#')
            class_id = act.ClassificationId(class1,class2.classificationItemId)
            elem_class = act.ElementClassification(element2.elementId,class_id)
            print("#Luokka#", Elementtityyppi,Tasotieto)

    ###_4.4_###_Kattoikkunat_####
    if "Kattoikkuna"==Elementtityyppi:
        if ("#Taso3#" in Tasotieto):
            class2 = acu.FindClassificationItemInSystem('#Luokitusjärjestelmä#','#Luokka#')
            class_id = act.ClassificationId(class1,class2.classificationItemId)
            elem_class = act.ElementClassification(element2.elementId,class_id)
            print("#Luokka", Elementtityyppi,Tasotieto)

    elem_classes.append(elem_class)
    acc.SetClassificationsOfElements(elem_classes)

print("Luokitukset ajettu elementteihin. Luokittele manuaalisesti elementtien sisäiset
elementtiosat (Verhorakenne, kaide, porras)")
#_5_##########################OHJELMA PÄÄTTYY#############################
```