hyväksymispäivä arvosana

arvostelija

Sovelluskielen hyödyntäminen peliohjelmoinnissa

Sami Suuronen

Helsinki 6.4.2012

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta – Fakultet –Faculty	Laitos – Institution – Department
Matemaattis-luonnontieteellinen tiedekunta	Tietojenkäsittelytieteen laitos

Tekijä – Författare – Author

Sami Suuronen

Työn nimi – Arbetets titel – Title

Sovelluskielen hyödyntäminen peliohjelmoinnissa

Oppiaine – Läroämne – Subject

Tietojenkäsittelytiede

Työn laji – Arbetets art – Level	Aika – Datum – Month and year	Sivumäärä – Sidoantal – Number of pages
	6.4.2012	32 sivua + 0 liitesivu(a)

Tiivistelmä – Referat – Abstract

Yleisen tason ohjelmointikielet, kuten Java sekä C++, tarjoavat työkalut yleisen ohjelmointiongelman ratkaisuun. Joskus näistä ohjelmointikielistä kuitenkin puuttuvat täsmätyökalut jonkin erityisen ohjelmointiongelman kuvaamiseen ja ratkaisuun. Tämä johtaa väistämättä tarpeettoman suureen määrään koodia, jota on hankala lukea ja ylläpitää.

Ongelmaa on pyritty ratkaisemaan luomalla erityisiä aihepiirin semantiikan sisältäviä *sovelluskieliä* (domain-specific language), joista tarvittavat täsmätyökalut löytyvät jo valmiina. Sovelluskielet perustuvat yleiskieliä korkeamman tason abstraktioihin ja tuoterunkoarkkitehtuureihin, joissa on tunnistettu aihepiirille tyypillisiä käsittelytarpeita.

ACM Computing Classification System (CCS):

Avainsanat – Nyckelord – Keywords

sovelluskieli, peliohjelmointi

Säilytyspaikka – Förvaringställe – Where deposited

Muita tietoja – Övriga uppgifter – Additional information

1 Johdanto

Yleisen tason ohjelmointikielet (general purpose programming language), kuten Java sekä C++, tarjoavat työkalut minkä tahansa ohjelmointiongelman ratkaisuun. Joskus näistä ohjelmointikielistä kuitenkin puuttuvat täsmätyökalut jonkin erityisen ohjelmointiongelman kuvaamiseen ja ratkaisemiseen. Ohjelmointikielen käsitteet ovat siis semanttisesti kaukana aihepiirin käsitteistä. Tämä johtaa väistämättä tarpeettoman suureen määrään koodia, jolloin sen ymmärrettävyys sekä ylläpidettävyys kärsivät.

Yleisessä tapauksessa ohjelmointiongelman ratkaisu voidaan jakaa aihepiirikohtaiseen sekä sovelluskohtaiseen osaan. Ongelmaa on pyritty ratkaisemaan ottamalla aihepiirikohtainen osa kiinteäksi osaksi ratkaisua ja jättämällä sovelluskohtainen osa sovelluskohtaisen ratkaisun varaan. Tähän on ehdotettu esimerkiksi aihepiirikohtaisten kirjastojen, sovelluskehysten sekä sovelluskielten käyttöä.

Tässä tutkielmassa tarkastellaan erityisesti sovellusten tuottamista tiettyyn aihepiiriin sekä siihen käytettyjä menetelmiä. Yksittäisten komponenttien tai kirjastoresurssien luomiseen tähtäävät menetelmät eivät siis kuulu tarkastelun piiriin.

Kappaleessa 2 tarkastellaan aihepiirisuuntautuneen sovellustuotannon peruskäsitteitä, prosessia sekä menetelmiä. Kappaleessa 3 käydään läpi tyypillinen sovelluskielen elinkaari toteutuspäätöksestä toteutukseen. Kappaleessa 4 keskitytään sovelluskielen käytöllä saavutettaviin etuihin.

2 Aihepiirisuuntautunut sovellustuotanto

Perinteisessä sovellustuotannossa keskityttiin uusien ongelmien ratkaisemiseen [McI68]. Aiempia ratkaisuja ei siis pyritty systemaattisesti hyödyntämään vaan jokainen sovellus toteutettiin aina puhtaalta pöydältä. Tämä johti samankaltaisten ongelmien toistuvaan ratkaisemiseen eri sovelluksissa.

Yleisen ongelman ratkaisussa hyödynnetään usein aiempaa tietämystä. Sovellustuotannon tapauksessa olemassa oleviin sovelluksiin on sitoutunut toistuvien ongelmien ratkaisemisen yhteydessä kerättyä tietämystä. Sovellusten kasvaessa ja monimutkaistuessa kiinnostus aiempien ratkaisujen uusiokäyttöön on kasvanut.

2.1 Peruskäsitteet ja prosessi

Aihepiiri (domain) on erityinen ongelma- tai tehtäväkenttä, johon kehitetään useita samankaltaisia, mutta eri asiakkaiden vaatimukset täyttäviä sovelluksia [TTC95]. Se voidaan määritellä kaikkien siinä esiintyvien sovellusten ratkaisemien yhteisten ongelmien tai yhteisen toiminnallisuuden kautta [Tra94]. Vaihtoehtoisesti aihepiiri voidaan myös määritellä sen käsitteiden ja ongelmien kuvaamiseen käytetyn kielen kautta [Tra94] [SCK96].

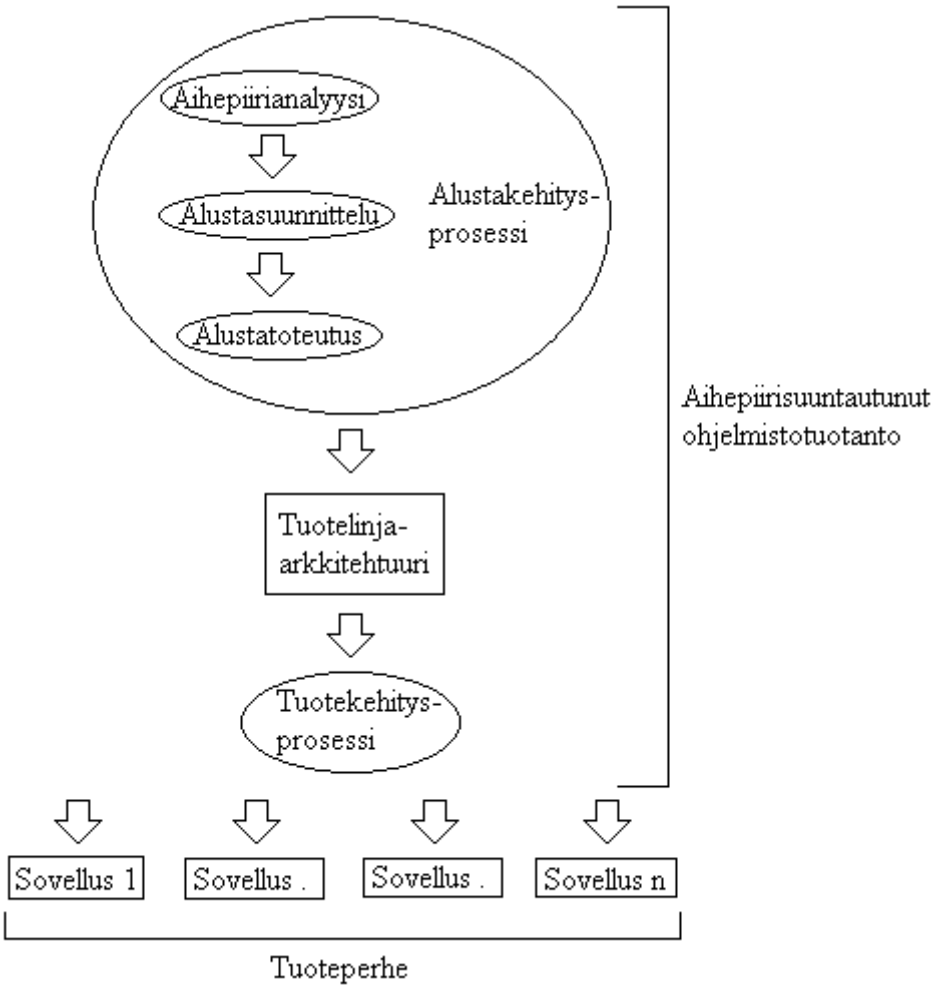
Aihepiirikohtainen sovellustuotanto perustuu sovellusten vaatimusten jakamiseen *aihepiirikohtaisiin vaatimuksiin* sekä *sovelluskohtaisiin vaatimuksiin*.

Aihepiirikohtaisiin vaatimuksiin on sitoutunut *aihepiiritietoa* (domain knowledge). Aihepiirikohtaiset vaatimukset liittyvät kyseisen aihepiiriin yhteisiin käsitteisiin sekä operaatioihin [Tra94], joten ne toistuvat jokaisessa aihepiirin sovelluksessa. Ne myös määrittävät aihepiirin [Tra94].

Sovelluskohtaiset vaatimukset puolestaan koskevat yksittäisiä sovelluksia. Ne eivät toistu systemaattisesti jokaisessa aihealueen sovelluksessa vaan vaihtelevat sovelluksesta toiseen. Sovellukset eroavat toisistaan juuri sovelluskohtaisten vaatimusten osalta.

Aihepiirikohtaiset vaatimukset toistuvat jokaisessa aihepiirin sovelluksessa, joten niiden toteutus voidaan ottaa kiinteäksi osaksi ratkaisumallia. Tällöin vain sovelluskohtaiset vaatimukset jäävät sovelluskohtaisen ratkaisun varaan. Uutta sovellusta toteutettaessa tarvitsee siis keskittyä vain sovelluskohtaisiin vaatimuksiin. Näin menettelemällä saadaan kätevästi uudelleenkäytettyä aiempia ratkaisuita sekä niihin sitoutunutta aihepiiritietoa.

Tracz [Tra94] sekä kuva 1 esittelevät *aihepiirisuuntautuneen sovellustuotannon* (domain-specific software engineering) keskeiset vaiheet ja käsitteet. Aihepiirisuuntautuneessa sovellustuotannossa keskitytään tuottamaan sovelluksia tiettyyn, hyvin rajattuun aihepiiriin. Aihepiirisuuntautuneessa sovellustuotannossa on tavoitteena hyödyntää systemaattisesti olemassa oleviin aihepiirin sovelluksiin sitoutunutta aihepiiritietoa.



Kuva 1. Aihepiirisuuntautuneen sovellustuotannon prosessi.

Aihepiirisuuntautunut sovellustuotanto voidaan jakaa kuvan 1 osoittamalla tavalla kahteen päävaiheeseen: *alustakehitysprosessiin* (domain engineering) sekä *tuotekehitysprosessiin* (application engineering).

Näistä kahdesta vaiheesta alustakehitysprosessi on selvästi laajempi. Alustakehitysprosessi voidaan edelleen jakaa kolmeen alivaiheeseen: *aihepiirianalyysiin* (domain analysis), *alustasuunnitteluun* (domain design), sekä *alustatoteutukseen* (domain implementation). Alustakehitysprosessin aikana tuotetaan aihepiirin sovellusten yhteisiin piirteisiin keskittyvä *tuotelinja-arkkitehtuuri* (domain-specific software architecture). Tuotelinja-arkkitehtuuri on nimestään huolimatta prosessi, jonka tarkoituksena on ohjata ja tukea sovellustuotantoa. Se käsittää aihepiirin sovellusten *viitearkkitehtuurin* (reference architecture) luomisen lisäksi esimerkiksi sovellusten tuottamiseen tarvittavan prosessin luomisen. Alustakehitysprosessin lopputuloksena saatava tuotelinja-arkkitehtuuri toimii

syötteenä seuraavalle vaiheelle.

Tuotekehitysprosessissa tuotetaan sovelluksia edellisessä vaiheessa tuotetun tuotelinja-arkkitehtuurin mukaisesti viitearkkitehtuuria ilmentämällä, erikoistamalla sekä laajentamalla. Tuotetut sovellukset yhdessä muodostavat *tuoteperheen* (product family). Jokaisella tuoteperheen jäsenellä on oma aihepiirin sovellusten viitearkkitehtuuriin pohjautuva *sovellusarkkitehtuurinsa* (application architecture).

Aihepiirisuuntautunut sovellustuotanto perustuu laajalti uusiokäyttöön, jonka avulla pyritään vähentämään sovellusten kehitysaikaa sekä kehityskustannuksia [TTC95]. Sen on tarkoitus myös parantaa niiden laatua, hallittavuutta sekä valmiutta uuteen liiketoimintaan [TTC95]. Edellä mainittujen etujen saavuttaminen ei kuitenkaan ole itsestään selvää. Aihepiiri on sekä määritettävä tarkasti että tunnettava perusteellisesti, jotta tämä lähestymistapa olisi kannattava [TTC95].

Tuotelinja-arkkitehtuuria ei siis kannata kehittää mille tahansa aihepiirille. Simos, Creps, Klinger ja kumppanit [SCK96] kiteyttävät yleisesti aihepiirille asetettavat vaatimukset seuraavasti. Aihepiirin on oltava kypsä eli siihen on oltava olemassa olevia sovelluksia. Aihepiirin tulee olla vakiintunut sekä sen sovellusten tyydyttäviä. Aihepiirin on lisäksi oltava taloudellisesti kannattava eli sille on oltava odotettavissa uusia sovelluksia. On esitetty, että tuotelinja-arkkitehtuurin pohjalta olisi kehitettävä vähintään kolme sovellusta, jotta sen tuottamisesta aiheutuneet kustannukset saataisiin kuoletettua [Tra94]. Kehittyneiden sekä laajalti käytettyjen kirjastojen olemassaolo viittaa aihepiirin olevan riittävän kypsä tuotelinja-arkkitehtuurien käytölle [TTC95].

Tämän kappaleen loppuosa käsittelee alustakehitysprosessia: 2.2 aihepiirianalyysi, 2.3 alustasuunnittelu sekä 2.4-2.6 alustatoteutukseen käytetyt menetelmät. Kappaleet 5, 6 ja 7 keskittyvät tuotekehitysprosessiin.

2.2 Aihepiirianalyysi

Aihepiirianalyysin ja aihepiiritiedon määritelmät

Neighbors [Nei80] esittelee termin *aihepiirianalyysi* (domain analysis). Tämän määritelmän mukaan aihepiirianalyysi keskittyy kaikkien tietyn aihepiirin sovellusten operaatioiden sekä olioiden tunnistamiseen. Aihepiirianalyysin tavoitteena on tukea aihepiirin uusien sovellusten luomista [Pri90]. Siinä missä *vaatimusanalyysi* (requirements ana-

lysis) tehdään yhtä sovellusta varten, aihepiirianalyysi tehdään kokonaista tuoteperhettä varten [Nei80][KCH90].

Aihepiirianalyysin aikana kerätään aihepiirille ominaista, aihepiirikohtaiset vaatimukset sisältävää *aihepiiritietoa* (domain knowledge). Aihepiiritiedon kerääminen on kuitenkin hankalaa, sillä se on usein epäformaalia, implisiittistä, erikoistunutta sekä epätäydellisesti mallinnettua ja sijaitsee yleensä vain aihepiirin asiantuntijoiden mielessä [IWA91]. Tämän lisäksi aihepiirin asiantuntijat ovat usein siinä määrin tottuneita niin aihepiiriin kuin sen käsitteisiin, että he olettavat aihepiiritiedon kuuluvan yleistiedon piiriin [TTC95]. Aihepiirianalyysin aikana kartoitettu aihepiiritieto tallennetaan vaiheen lopputuloksena saatavaan *aihepiirimalliin* (domain model) [Nei80] [Pri90] [KCH90] [Tra94] [MHS05].

Aihepiirianalyysiin on sittemmin kehitetty lukuisia erilaisia menetelmiä, kuten *Domain Analysis and Reuse Environment (DARE)* [FPF98], *Family-Oriented Abstractions, Specification, and Translation (FAST)* [CHW98], *Feature-Oriented Domain Analysis (FODA)* [KCH90], *Organization Domain Modeling (ODM)* [SCK96] sekä *Ontology-based Domain Engineering (ODE)* [AGK02].

Mernikin, Heeringin ja Sloanen [MHS05] mukaan nykyisin käytössä olevista aihepiirianalyysimenetelmistä on tunnistettavissa kolme yhteistä vaihetta. Rajataan aihepiiri. Kerätään aihepiiritietoa useista eri lähteistä (tekniset dokumentit, aihepiirin asiantuntijat, olemassa olevien sovellusten koodit sekä asiakaskyselyt). Muodostetaan aihepiirimalli aihepiirin määrittelystä, aihepiirin terminologiasta sekä aihepiirin käsitteiden kuvauksista. Aihepiirimalli sisältää myös aihepiirin käsitteiden yhteneväisyydet, eroavaisuudet sekä keskinäiset riippuvuudet sisältävät piirremallit.

Edellisessä luvussa esiteltyjen aihepiirikohtaisten vaatimusten määrä voidaan teoriassa maksimoida sekä sovelluskohtaisten vaatimusten määrä minimoida rajaamalla aihepiiri mahdollisimman tarkasti aihepiirianalyysin yhteydessä. Aihepiirin rajaaminen ei kuitenkaan ole aivan yksinkertaista [LaM97]. Tarkasteltaessa esimerkiksi joukkoa verkkosovelluksia, voidaan havaita niissä toistuvan jonkin verran samoja vaatimuksia. Aihepiirin rajausta tarkennettaessa esimerkiksi verkkosähköpostisovelluksiin, voidaan sovellusten aihepiirikohtaisten vaatimusten lukumäärän havaita kasvaneen selvästi. Aihepiiriä kavennettaessa tarkasteltavien sovellusten lukumäärä on kuitenkin reilusti vähentynyt.

On selvästi nähtävissä, että aihepiirin rajausta tarkennettaessa sovellusten yhteisten, aihepiirikohtaisten vaatimusten lukumäärä kasvaa, mutta tarkasteltavien sovellusten lukumäärä sitä vastoin vähenee. Tämän seurauksena aihepiiriarkkitehtuurin avulla tuotettavissa olevien sovellusten kirjo kapenee ja sen hyödyllisyys vähenee.

2.3 Alustasuunnittelu

Alustasuunnittelu (domain engineering) on prosessi, jossa kehitetään tuotelinja-arkkitehtuuri [Tra94] [TTC95]. Tuotelinja-arkkitehtuuri keskittyy aihepiirin ongelmien ratkaisemiseen ja sen kehittäminen edellyttää yhteistyötä järjestelmä- ja sovellussuunnittelijoiden kanssa [TTC95].

Taylor, Tracz ja Coglianese [TTC95] kuvaavat alustasuunnittelun päävaiheet. Alustasuunnittelussa kehitetään aihepiirianalyysivaiheessa kerätyn tiedon pohjalta geneerisiä arkkitehtuureja sekä määritellään niiden sisältämien moduulien tai komponenttien syntaksi ja semantiikka. Useamman arkkitehtuurin luominen samalle aihepiirille voi olla tarpeen aihepiirin rajoitteista riippuen.

Alustasuunnitteluprosessi jaetaan neljään päävaiheeseen. Ensimmäisessä vaiheessa määritellään tuotelinja-arkkitehtuuri, joka jaetaan kolmeen vaiheeseen. Ensin kehitetään aihepiirianalyysivaiheessa tunnistettuihin rajoitteisiin perustuva korkean tason suunnitelma tai suunnitelmat. Seuraavaksi tunnistetaan korkean tason suunnitelmista korkean tason komponentit. Lopuksi kehitetään prosessi tuotelinja-arkkitehtuurin konfiguroimiseen erityisten vaatimusten ja rajoitusten mukaisesti. Perustelut ja valinnat tallennetaan ja kehityksessä keskitytään aihepiirin tärkeimpien abstraktioiden tunnistamiseen.

Alustasuunnitteluprosessin toisessa vaiheessa määritellään moduulit, joiden avulla geneerisestä suunnitelmasta voidaan tuottaa tiettyjä ilmentymiä. Moduulien määritelmien tulee määritellä syntaktiset sekä semanttiset rajapinnat, suorituskyky- ja ajoitusominaisuudet, riippuvuudet toisista moduuleista, kontrollisuhteet toisiin moduuleihin sekä moduulin käytöstä seuraavat rajoitteet. Ratkaisuiden tulee tukea mahdollisimman laajaa uusiokäyttöisyyttä. Suunnittelussa tulee myös määrittää miten moduulin parametroinnilla voidaan parantaa sen yleiskäyttöisyyttä aihepiirin rajoitteet huomioon ottaen. Kaikki suunnittelupäätökset perusteineen tallennetaan. Ensimmäinen ja toinen vaihe ovat iteratiivisia.

Alustasuunnitteluprosessin kolmannessa vaiheessa edellisessä vaiheessa suunniteltuihin moduuleihin tulee liittää viittaukset edellisen vaiheen käsitteisiin ja vaatimuksiin. Viittauksista tulee käydä ilmi kunkin moduulin täyttämät vaatimukset sekä aihepiiristä mallinnettu osuus. Neljännessä ja viimeisessä vaiheessa iteroidaan edellisiä vaiheita abstraktiotasoa samalla laskien kunnes riittävän yksityiskohtainen taso on saavutettu.

Alustasuunnitteluprosessin aikana tuotettu korkean tason suunnitelma tai suunnitelmat täytetään alustatoteutusvaiheessa aihepiirin uusien sovellusten tuottamiseen käytettävillä konkreettisilla komponenteilla. Seuraavissa kolmessa kappaleessa esitellään alustatoteutukseen kolme vaihtoehtoista menetelmää: kirjastot, sovelluskehykset sekä sovelluskielet.

2.4 Alustatoteutus: kirjastot

Ohjelmakirjasto (library) on kokoelma resursseja. Toistuvan vaatimuksen toteuttavan resurssin sijoittaminen ohjelmakirjastoon on klassinen tapa pakata uudelleenkäytettävää koodia sekä aihepiiritietoa [DKV00] [TTC95]. Moore [Moo61] kuvaa yleisesti käytettyjen aliohjelmien sijoittamisen ohjelmakirjastoon, josta niitä voidaan tarvittaessa kutsua. McIlroy [McI68] vie ideaa vielä pidemmälle ehdottamalla yleiskäyttöisten, standardoitujen ohjelmakirjastojen tuomista kiinteäksi osaksi sovellustuotantoa. Tässä lähestymistavassa kehittäjät rakensivat sovelluksia standardikirjastojen sisältämistä uudelleenkäytettävistä komponenteista [KMB96] [Kru92].

McIlroy:n lähestymistavassa uudelleenkäytettävien komponenttien tyyppi rajoittui kuitenkin lähinnä toiminnallisuuteen [Kru92]. Sittemmin ohjelmointikielten kehittymisen myötä uudelleenkäytettävien komponenttien tyyppi on laajentunut toiminnallisuudesta tietotyyppeihin [Kru92]. Konkreettisella tasolla on siirrytty aliohjelmista moduuleihin, paketteihin sekä luokkiin [Kru92]. Esimerkki yleiskäyttöisestä sekä usein tarvitusta kirjastoresurssista on Java:n java.lang-paketin sisältämä String-luokka [GJS05].

Yleisen tason ohjelmointikieltä voidaan siis laajentaa luomalla erityinen aihepiirin semantiikan sisältävä kirjasto [MHS05]. Tässä tapauksessa kirjaston *ohjelmointirajapinta* (Application Programmer's Interface, API) muodostaa aihepiirikohtaisen sanaston sisältämiensä luokkien, metodien ja funktioiden nimistä [MHS05]. Kontrolli on kutsuvalla sovelluksella, joka voi vapaasti käyttää aihepiirikohtaisia kirjastoresursseja [DKV00] [Joh97].

Kirjastokomponenttien tarkoitus on helpottaa uuden sovelluksen tuottamista jakamalla kirjastokomponentin toteuttamisesta aiheutuneet kertaluontoiset kustannukset kaikkien sen käyttökertojen kesken [Kru92]. Komponentit on kuitenkin löydettävä ennen kuin niitä voidaan käyttää. Tätä varten komponenttikirjaston on tarjottava tehokas luokitteluja hakumenetelmä komponenttien etsintään [Kru92].

Kirjastoresurssien käyttö tukee sekä koodin että aihepiirianalyysin uusiokäyttöä. Koodin uusiokäyttö perustuu kirjastoresurssien sisältämän koodin toistumiseen kaikissa sitä käyttävissä sovelluksissa. Aihepiirianalyysin uusiokäyttö perustuu aihepiirin käsitteiden ja sanaston toistumiseen kaikissa kirjastoresurssia käyttävissä sovelluksissa.

Kirjastoresurssien käytöllä saavutetaan kaksi merkittävää etua. Ensinnäkin kirjastoresursseilla toteutettavien osien käytännön toteutus saadaan kätevästi abstrahoitua pois. Toiseksi kirjastojen käyttö on usein kustannustehokkain keino resurssien uudelleenkäyttöön [MHS05].

Kirjastokomponenttien laajamittainen hyödyntäminen aihepiirikohtaisessa sovelluskehityksessä on kuitenkin haastavaa. Ensiksikin ytimekkään, komponenttia tarkasti kuvaavan abstraktion löytäminen on vaikeaa [Kru92]. Toiseksi, aihepiirikohtaisten käsitteiden ja abstraktioiden kuvaaminen kirjastoresursseilla ei aina ole suoraviivaista: joskus yleisen tason ohjelmointikieli ei onnistu aihepiirikohtaisen kirjastonkaan kanssa luontevasti kuvaamaan aihepiirin käsitteitä ja ongelmia [MHS05]. Kolmanneksi [TTC95], useimmat kirjastot sisältävät matalan tason komponentteja, jotka eivät riittävässä määrin tue sovellustuotantoa. Tyypillisesti näiden komponenttien lisäksi joudutaan kirjoittamaan suuret määrät koodia sovelluksen luomiseksi. Lisäksi kirjastojen sisältämien komponenttien hyödyllisyyden hahmottaminen saattaa olla hankalaa. Neljänneksi, kirjastojen kehittäminen on kallista ja siitä saatava hyöty kyseenalaista [CHW98].

2.5 Alustatoteutus: sovelluskehykset

Beckin ja Johnsonin [BeJ94] määritelmän mukaan *sovelluskehys* (application framework) on uusiokäyttöinen, joukosta abstrakteja luokkia sekä niiden välisistä suhteista koostuva järjestelmän tai sen osan malli.

Sovelluskehys pyrkii edistämään aihepiirikohtaisten, hyväksi havaittujen sovellusmallien sekä niiden toteutusten uusiokäyttöä [FaS97] [FHL97] [Joh97]. Aihepiirikohtaiset

vaatimukset toteuttavia resursseja ei tarvitse sijoittaa yleiseen kirjastoon vaan ne voidaan ottaa osaksi sovelluskehystä [FHL97] [DKV00]. Kirjastoista poiketen sovelluskehykset ovat usein aihepiirikohtaisia [FaS97].

Abstraktit luokat ovat tärkeässä roolissa sillä ne toimivat rajapintoina mahdollistaen komponenttien käytännön toteutuksen muuttamisen [Joh97]. Sovelluskehys kuvaa miten sovellus voidaan jakaa olioihin [Joh97]. Sovelluskehyksen tärkein osa on kontrollin kulku olioiden välillä sekä niiden vuorovaikutusmalli [Joh97].

Sovelluskehykset voidaan luokitella abstraktiuden perusteella täysin abstrakteihin sekä osittain abstrakteihin. Täysin abstrakti sovelluskehys määrittää tuotettavan sovelluksen rakenteen, mutta ei ota kantaa sen toiminnallisuuteen eikä kontrollin kulkuun. Osittain abstraktit sovelluskehykset määrittävät rajapinnat sekä kontrollin kulun. Lisäksi ne sisältävät aihepiirikohtaisia vaatimuksia täyttäviä konkreettisia luokkia.

Fayad ja Schmidt [FaS97] kuvaavat osittain abstrakteja sovelluskehystyyppejä. Osittain abstraktit sovelluskehykset voidaan edelleen jakaa laajennusmekanismin perusteella *muunneltaviin kehyksiin* (white-box framework) sekä *koottaviin kehyksiin* (black-box framework). Muunneltavan sovelluskehyksen toiminnallisuuden laajentaminen perustuu oliomenetelmiin, kuten perintään sekä dynaamiseen sidontaan. Muunneltavan kehyksen käyttö kuitenkin edellyttää sen sisäisen rakenteen tarkkaa tuntemista. Muunneltavan kehyksen avulla tapahtuvassa sovelluskehityksessä sovelluskehyksen toiminnallisuutta laajennetaan joko sovelluskehyksen luokkia perimällä tai sovelluskehyksen metodeja syrjäyttämällä. Koottavan sovelluskehyksen toiminnallisuuden laajentaminen perustuu sovelluskehykseen liitettävien komponenttien rajapintojen määrittämiseen. Toiminnallisuuden laajentaminen on helpompaa kuin muunneltavassa kehyksessä ja se tapahtuu rajapintamäärittelyt täyttäviä komponentteja kehittämällä ja niitä sovelluskehykseen liittämällä.

Froehlich, Hoover, Liu ja kumppanit [FHL97] kuvaavat sovelluskehystä hyödyntävää tuotekehitysprosessia seuraavasti. Sovelluskehyksestä johdetaan sovelluksia erikoistamalla sekä laajentamalla tiettyjä sovelluskehyksen osia. Erityisten *laajennuskohtien* (hook) avulla sovelluskehyksen kehittäjä voi viestiä sovelluskehittäjille mitkä kohdat on erikoistettava, jotta sovelluskehyksestä voidaan johtaa sovelluksia. Erikoistamisen yhteydessä sovelluskehyksen määrittämä rakenne ei muutu, joten saman sovelluskehyksen

pohjalta kehitetyillä sovelluksilla on sama rakenne.

Kirjastoista poiketen sovelluskehykset käyttävät *käänteistä kutsurakennetta* (inversion of control) [FaS97] [DKV00] [Joh97]. Käänteisessä kutsurakenteessa kontrolli on sovelluskehyksellä, joka käyttää laajennuskohtiin sijoitettuja, sovelluskohtaisia resursseja [FaS97] [DKV00] [Joh97].

Sovelluskehys tukee koodin, rakenteen sekä aihepiirianalyysin uusiokäyttöä. Koodin uusiokäyttö perustuu sovelluskehyksen sisältämien konkreettisten luokkien toistumiseen kaikissa siitä johdetuissa sovelluksissa, rakenteen uusiokäyttö perustuu sovelluskehyksen rakenteen toistumiseen kaikissa siitä johdetuissa sovelluksissa ja aihepiirianalyysin uusiokäyttö perustuu aihepiirin käsitteiden ja sanaston toistumiseen kaikissa sovelluskehyksestä johdetuissa sovelluksissa [Joh97].

On tärkeää huomata, että sovelluskehyksen sisältämien luokkien uusiokäyttö tapahtuu niiden alkuperäisessä kontekstissa, joten niitä ei tarvitse modifioida. Lisäksi sovelluskehykset ratkaisevat suurempia ongelmia kuin yksittäiset kirjastoresurssit, minkä vuoksi niiden etsiminen ja uusiokäyttö on huomattavan kustannustehokasta yksittäisiin kirjastoresursseihin verrattuna [FHL97]. Hyvä esimerkki sovelluskehyksestä on php-verkkosovellusten kehittämiseen tarkoitettu Zend Framework.

Fayad ja Schmidt [FaS97] tunnistavat tärkeimmiksi sovelluskehysten käytöllä saavutettaviksi hyödyiksi parantuneen modulaarisuuden, uudelleenkäytettävyyden, laajennettavuuden sekä *käänteisen kutsurakenteen* (inversion of control). Modulaarisuutta saadaan edistettyä kapseloimalla sovelluksittain vaihtelevat osat rajapintojen taakse. Tämä parantaa sovelluksen laatua estämällä tietyn moduulin rakenteeseen ja toteutukseen tehtävien muutosten heijastumista muihin moduuleihin. Uudelleenkäytettävyyttä voidaan parantaa määrittelemällä rajapinnoissa uudelleenkäytettävät komponentit. Sovelluskehyksen uudelleenkäytettävyys parantaa aihepiiritietoa sekä vähentää tarvetta toistuvasti ratkaista samoja ongelmia. Lisäksi se voi parantaa tuottavuutta, laatua, suorituskykyä, luotettavuutta sekä yhteentoimivuutta. Laajennettavuutta voidaan parantaa tarjoamalla *koukkumetodeja* (hook methods), joilla sovelluskehyksen rajapintoja voidaan laajentaa. Koukkumetodit erottavat järjestelmällisesti vakaat rajapinnat sovelluskohtaisista osista. Käänteinen kutsurakenne on tunnusomaista sovelluskehyksen suoritusaikaiselle arkkitehtuurille. Suoritus on *tapahtumaohjattua* (event driven). *Tapahtuman* (event) sattuessa

lähettäjä (dispatcher) siirtää kontrollin kyseisestä tapahtumasta vastaavalle *käsittelijäoliolle* (event handler object). Suurin osa sovelluskohtaisesta toiminnallisuudesta sijoitetaan juuri käsittelijäolioihin niitä laajentamalla ja erikoistamalla. Sovelluskehykset voivat myös merkittävästi parantaa sovelluksen laatua sekä vähentää kehityskustannuksia [FHL97].

Jotta aiemmin lueteltuihin hyötyihin päästäisiin käsiksi, sovelluskehystä on käytettävä tarkoituksenmukaisesti. Sovelluskehyksen ymmärtäminen saattaa kuitenkin olla vaikeaa sen monimutkaisuuden vuoksi [FHL97] [Joh97]. Fayad ja Schmidt [FaS97] kuvaavat sovelluskehyksen käytöstä aiheutuvia vaikeuksia seuraavasti. Laadukkaan, laajennettavan ja uusiokäyttöisen sovelluskehyksen kehittäminen on erittäin vaikeaa. Lisäksi sovelluskehyksen tehokkaan käytön oppimiseen kuluu paljon aikaa. Käytännössä samaa sovelluskehystä on käytettävä useassa projektissa ennen kuin sen käyttö tulee taloudellisesti kannattavaksi.

2.6 Alustatoteutus: sovelluskielet

Deursenin, Klintin ja Visserin [DKV00] sekä Mernikin, Heeringin ja Sloanen [MHS05] määritelmiä yhdistämällä *sovelluskieli* (domain-specific language) on ohjelmointikieli tai määrittelykieli, joka tarjoaa sopivien notaatioiden ja abstraktioiden kautta tiettyyn aihepiiriin keskittynyttä ja yleensä myös rajoittunutta ilmaisuvoimaa.

Käytännössä luokittelu sovelluskieleksi on kuitenkin monimutkaisempaa, mistä johtuen useat ohjelmointikielet voidaan tulkinnasta riippuen mieltää joko yleisen tason ohjelmointikieliksi tai sovelluskieliksi [MHS05]. Ohjelmointikielten evoluutio vaikeuttaa rajanvetoa entisestään. Osaa alun perin sovelluskieliksi mielletyistä ohjelmointikielistä on laajennettu niin paljon että niitä pidetään nykyään yleisen tason ohjelmointikielinä [DKV00].

Sovelluskieli sisältää aihepiirin semantiikan. Näin ollen sitä voidaan pitää korkeimman asteen abstraktiona, sillä se sisältää vain ongelman ratkaisun kannalta merkitykselliset piirteet [Hud96]. Sovelluskielten avulla sovellustuotantoa tuntematon käyttäjä voi kuvata aihepiirin ongelmia luontevasti aihepiirin termien ja käsitteiden avulla [Bar85]. Kuvauksen ei myöskään tarvitse olla muodollinen, tarkka tai yksityiskohtainen [Bar85]. Tämä mahdollistaa jopa loppukäyttäjän suorittaman sovelluskehityksen [HKN85].

Sovelluskielen idea ei suinkaan ole uusi. Ensimmäisenä sovelluskielenä voidaan pitää viisikymmentäluvun puolivälissä kehitettyä, työkoneiden numeeriseen ohjaamiseen tarkoitettua APT:tä [HMS03]. Tämän jälkeen on kehitetty satoja erilaisia sovelluskieliä, joista kuitenkin vain osa on päätynyt kirjallisuuteen [DKV00].

Sovelluskielet voidaan toteutustavan perusteella jakaa kahteen ryhmään. Hudak [Hud96] esittelee *sulautetun sovelluskielen* (domain-specific embedded language) idean. Tässä lähestymistavassa sovelluskieltä ei lähdetä toteuttamaan puhtaalta pöydältä vaan aihepiirin semantiikka rakennetaan yleisen tason ohjelmointikieleen. Sulautettu sovelluskieli ei siis esiinny itsenäisesti vaan yleisen tason ohjelmointikielen laajennuksena. Sulautettu sovelluskieli kasvattaa yleisen tason ohjelmointikielen ilmaisuvoimaa tuomalla aihepiirikohtaisen semantiikan osaksi sitä. Sulautetun sovelluskielen tapauksessa hyödynnetään alustana toimivan ohjelmointikielen omaa kääntäjää eikä erilliselle kääntäjälle ole tarvetta.

Loput sovelluskielet ovat *ulkoisia sovelluskieliä* (domain-specific external language). Ulkoiset sovelluskielet tarvitsevat oman kääntäjän tai tulkin. Ulkoisella sovelluskielellä laaditusta ohjelmasta tai sen määrittelystä saadaan tuotettua sovellus sitä varten kehitetyn kääntäjän eli *sovellusgeneraattorin* (application generator) avulla [Cle88] [DKV00] [SMT09]. Sovellusgeneraattorin syötteet ovat kuitenkin perinteisestä kääntäjästä poiketen aihepiirikohtaisia, korkean tason abstraktioita [Kru92]. Sovellusgeneraattorien käyttö tukee vahvasti tuotelinja-arkkitehtuuriajattelua, sillä sovellusgeneraattorin syöte voidaan nähdä tuotelinja-arkkitehtuuriin perustuvan sovellusarkkitehtuurin formaalina esityksenä [TTC95]. Lisäksi aihepiirikohtaiset vaatimukset voidaan ottaa osaksi kääntäjää ja jättää sovelluskohtaiset vaatimukset sovelluskielen avulla toteutettaviksi [Bar85] [Cle88] [Kru92]. Näin toimimalla sovelluksen määrittely saadaan erotettua sen toteutuksesta [Bar85] [Cle88].

Bentley [Ben86] esittelee *pienoissovelluskielen* (little language) käsitteen. Tämän määritelmän mukaan pienoissovelluskieli on tiettyyn aihepiiriin suunniteltu, pieni sovelluskieli joka sisältää vain harvoja perinteisissä yleistason ohjelmointikielissä esiintyviä piirteitä.

Useat sovelluskielet ovat luonteeltaan kuvailevia, minkä vuoksi ne voidaan nähdä määrittelykielinä [DKV00]. Cleveland [Cle88] kuvaa *määrittelysovelluskielten* (application-

oriented language, specification language) käyttöä sovellusten tuottamisessa. Kuvauksen mukaan määrittelysovelluskieli voi olla joko vuorovaikutteisen valikon, graafisen kaavion tai kirjoitetun kielen muodossa.

Neljännen sukupolven ohjelmointikielet (4th generation languages) ovat yleensä yritystietojärjestelmien kehittämiseen suunnattuja sovelluskieliä [DKV00].

Toteutustavan lisäksi sovelluskielet voidaan jakaa myös käyttäjän perusteella kahteen ryhmään. Kehittäjille suunnatuissa sovelluskielissä ohjelmoinnin suorittaa kehittäjä. Loppukäyttäjille suunnatuissa sovelluskielissä ohjelmoinnin voi suorittaa varsinaista ohjelmointia taitamaton henkilö, esimerkiksi aihepiirin asiantuntija [SuM04]. Ohjelmointiin pätevät kuitenkin samat lainalaisuudet ohjelmointikielen tasosta riippumatta, mistä johtuen monet aihepiirin asiantuntijat saattavat kokea sovelluskielelläkin ohjelmoinnin haastavaksi [SmB83]. Lisäksi loppukäyttäjän kirjoittamissa ohjelmissa piilee oma vaaransa [Har04].

Sovelluskieli tukee rakenteen, koodin sekä aihepiirianalyysin uusiokäyttöä [MHS05] [Kru92]. Rakenteen uusiokäyttö perustuu sovelluskielellä kirjoitetun ohjelman rakenteen toistumiseen kaikissa sovelluskielellä kirjoitetuissa sovelluksissa. Koodin uusiokäyttö perustuu kirjastoresursseja hyödyntävissä sovelluskielissä kirjastoresurssien toistumiseen kaikissa sovelluskielellä kirjoitetuissa sovelluksissa. Aihepiirianalyysin uusiokäyttö perustuu aihepiirin käsitteiden ja sanaston toistumiseen kaikissa sovelluskielellä kirjoitetuissa sovelluksissa.

Sovelluskielen käytöllä voidaan teoriassa saavuttaa monenlaisia hyötyjä yleistason ohjelmointikieliin verrattuna. Sovelluskieli mahdollistaa aihepiirin käsitteillä tapahtuvan ohjelmoinnin [Bar85] [MMS05] [SMT09]. Suuremman ilmaisuvoiman ansiosta koodin määrä vähenee, tuottavuuden, luotettavuuden, uudelleenkäytettävyyden, ylläpidettävyyden ja verifioitavuuden parantuessa [HMS03]. Formaalien metodien näkökulmasta on tärkeää huomata, että kaikki päättely ja todistelu voidaan tehdä aihepiirin käsitteillä [Hud96] [Bas97] [MMS05]. Sovelluskielen käyttö edistää laajamittaista uusiokäyttöä [MHS05]. Lisäksi kehitysaika lyhenee ja sitä myötä myös kehityskustannukset pienenevät huomattavasti [MHS05] [SMT09]. Haittapuolena voidaan pitää näiden kielten hyödyttömyyttä oman aihepiirinsä ulkopuolella [MHS05].

Yleisesti tunnettuja esimerkkejä sovelluskielistä ovat esimerkiksi HTML [RLJ99] sekä SQL. Sovelluskieli voi myös olla suunnattu loppukäyttäjää varten. Hyvä esimerkki tästä on Excel-taulukkolaskentaohjelman kaavakieli [DKV00] [MHS05].

Yleisessä tapauksessa uuden sovelluskielen luominen kannattaa vain jos sen käytöstä saatava hyöty ylittää sen luomisesta aiheutuvat kustannukset [MHS05]. Uuden sovelluskielen luominen on aina työläs prosessi eikä sitä kannata lähteä suunnittelemaan ja toteuttamaan jokaista vastaantulevaa ongelmaa varten [MHS05]. Se on kannattavaa lähinnä tilanteessa, jossa saman ongelman odotetaan toistuvan riittävän usein samassa kontekstissa eikä sen ratkaisemiseen ole entuudestaan olemassa kunnollisia työkaluja.

3 Sovelluskielen elinkaari

Tyypillinen sovelluskielen elinkaari koostuu viidestä vaiheesta: toteutuspäätös, aihepiirianalyysi, suunnittelu, toteutus ja käyttö [MHS05]. Tässä kappaleessa keskitytään neljään ensin mainittuun.

3.1 Toteutuspäätös

Mernik, Heering ja Sloane [MHS05] kuvaavat toteutuspäätöstä seuraavasti. Uuden sovelluskielen toteuttamiselle on yleensä oltava painavat perustelut eikä siihen kannata lähteä jokaista vastaantulevaa ongelmaa varten. Kyseessä on työläs prosessi, joka edellyttää tietämystä sekä aihepiiristä että kielten kehittämisestä. Käytännössä aihepiirin ja kielten kehittämisen asiantuntijat ovat valitettavan harvassa. Lisäksi sovelluskielten toteuttamistekniikat ovat yleiskielten vastaavia monimutkaisemmat ja vaativat useiden tekijöiden tarkkaa huomioimista. Myös kohdeorganisaation koko saattaa asettaa omat haasteensa, koska koulutusmateriaalin, käyttötuen, standardoinnin sekä ylläpidon kehittäminen voi olla suuritöistä sekä aikaa vievää.

Yleisessä tapauksessa uuden sovelluskielen toteuttaminen kannattaa vain jos sen käytöstä saatavan hyödyn odotetaan ylittävän sen kehittämisestä ja käyttöönotosta aiheutuvat kustannukset [MHS05]. Toisin sanoen sovelluskielen toteuttaminen kannattaa lähinnä tilanteessa, jossa saman ongelman odotetaan toistuvan riittävän usein samassa kontekstissa eikä sen ratkaisemiseen ole entuudestaan olemassa kunnollisia työkaluja. Odotettu hyöty voi olla esimerkiksi alentuneiden sovelluskehitys- ja ylläpitokustannusten muodossa [MHS05]. Lisäksi uuden sovelluskielen kehittämisen tavoitteena voi olla sovel-

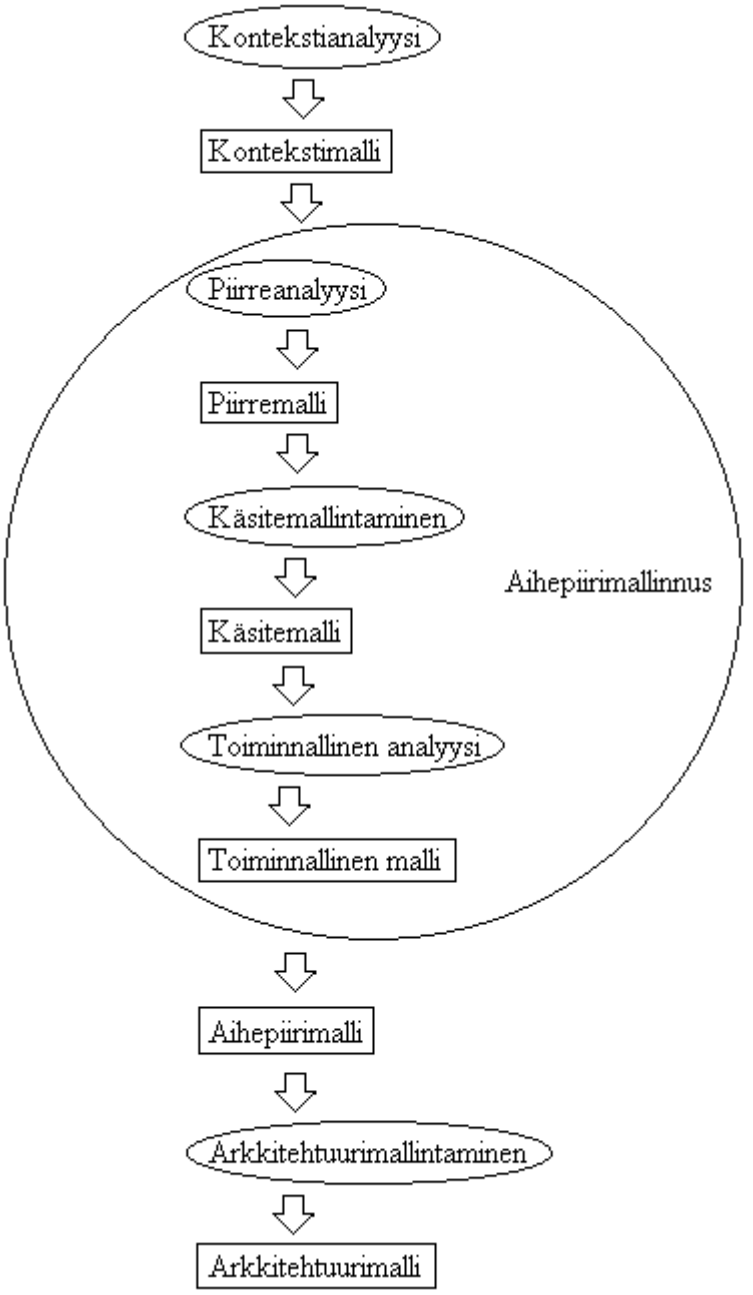
luskehityksen tuominen uusien käyttäjäryhmien ulottuville. Uusi sovelluskieli voi olla suunnattu aihepiirin asiantuntijoille tai loppukäyttäjille, joilla ei ole merkittävää aiempaa ohjelmointikokemusta [MHS05] [SuM04].

Simos, Creps, Klinger ja kumppanit [SCK96] esittävät kolme vaatimusta aihepiirille. Aihepiirin on oltava kypsä eli siihen on olemassa olevia sovelluksia. Aihepiirin on oltava vakiintunut ja sen sovellusten tyydyttäviä. Aihepiirin on myös oltava taloudellisesti kannattava eli sille on oltava odotettavissa uusia sovelluksia. Lisäksi kehittyneiden sekä käytettyjen kirjastojen olemassaolo viittaa aihepiirin olevan riittävän kypsä tuotelinja-arkkitehtuurien käytölle [TTC95].

3.2 Aihepiirinalyysi

Aihepiirianalyysiin on olemassa useita erilaisia tekniikoita. Seuraavassa tarkastellaan esimerkinomaisesti *piirresuuntautunutta aihepiirianalyysiä* (feature-oriented domain analysis, FODA).

Kang, Cohen, Hess ja kumppanit [KCH90] kuvaavat piirresuuntautuneen aihepiirianalyysin päävaiheiksi *kontekstianalyysin* (context analysis), *aihepiirimallinnuksen* (domain modelling) sekä *arkkitehtuurimallinnuksen* (architectural modelling). Jokaisessa vaiheessa tuotetaan malli, joka toimii seuraavan vaiheen syötteenä. Piirresuuntautuneen aihepiirianalyysin prosessi on esitetty kuvassa 2.



Kuva 2. Piirresuuntautunut aihepiirianalyysi.

Kontekstianalyysin (context analysis) tavoitteena on määritellä aihepiiri, josta todennäköisesti löytyy hyödynnettävissä olevia sovelluksia. Kontekstianalyysivaiheessa kartoitetaan aihepiirin ja ulkoisten elementtien väliset suhteet sekä niiden vaihtelu. Lisäksi kartoitetaan ulkoisten olosuhteiden, kuten toimintaympäristön vaihtelu. Kontekstianalyysin tuloksia hyödynnetään aihepiirin rajaamisessa. Aihepiirin rajauksessa otetaan huomioon aihepiirin sovellusten yhteiset piirteet, aihepiiritiedon saatavuus, aihepiirin sovellusten odotettu käyttö sekä projektin resurssit ja rajoitteet. Aihepiirissä esiintyvä

runsas vaihtelu, jota ei saada abstrahoitua pois saattaa olla merkki yhteisten tekijöiden puuttumisesta. Tällöin aihepiirin uudelleenrajaus voi olla tarpeen. Kontekstianalyysin tulokset tallennetaan *kontekstimalliin* (context model). Kontekstimalli määrittelee aihepiirin sekä sitä kautta myös koko aihepiirianalyysin rajat.

Aihepiirimallinnusvaiheessa kartoitetaan edellisessä vaiheessa rajatun aihepiirin sovellusten ratkaisemien ongelmien yhteneväisyydet ja eroavaisuudet. Aihepiirimallinnus voidaan edelleen jakaa kolmeen alivaiheeseen: *piirreanalyysiin* (feature analysis), *käsitemallintamiseen* (entity-relationship modelling) sekä *toiminnalliseen analyysiin* (functional analysis).

Piirreanalyysin tarkoituksena on kartoittaa käyttäjien näkemys aihepiirin sovellusten ominaisuuksista sekä löytää aihepiirin sovellusten aihepiirikohtaiset ja sovelluskohtaiset piirteet. Tarkastelun kohteena ovat erityisesti käyttäjälle näkyvät piirteet eli sovellusten tarjoamat palvelut sekä niiden toimintaympäristö.

Piirreanalyysissa tarvittavaa tietoa on saatavilla monesta lähteestä, joilla jokaisella on omat vahvuutensa ja heikkoutensa. Kirjat ovat hyvä aihepiiritiedon, teorian, metodien, tekniikoiden sekä mallien lähde. Toisaalta ne saattavat olla kirjoittajan omien mielipiteiden värittämiä. Lisäksi mallit saattavat olla yksipuolisia. Standardeista puolestaan löytyy aihepiirien viitemalleja, mutta nämä eivät välttämättä ole ajan tasalla. Olemassa olevat sovellukset ovat tärkein aihepiiritiedon lähde ja niitä voidaan suoraan käyttää käyttäjälle näkyvien piirteiden määrittämiseen. Sovellusten vaatimusdokumentteja voidaan myös käyttää aihepiiritiedon lähteenä, jonka lisäksi suunnittelumalli ja lähdekoodi kuvaavat arkkitehtuurin. Haittapuolena voidaan pitää useiden järjestelmien analysoimisesta syntyviä korkeita kustannuksia. Aihepiirin asiantuntijat voivat tarjota muuten vaikeasti löydettävää taustatietoa, toimia konsultteina aihepiirianalyysin aikana sekä myöhemmässä vaiheessa tarkastaa tuotettuja sovelluksia. Yleisessä tapauksessa jokainen asiantuntija on kuitenkin erikoistunut omaan kapeaan erikoisalaansa, mistä johtuen koko aihepiirin kattamiseen saatetaan tarvita useita asiantuntijoita.

Piirteiden tunnistamiseen tarvittavat tiedot löytyvät usein käyttöoppaista sekä vaatimusdokumenteista. Tunnistettavat piirteet ovat käyttäjälle näkyviä ja ne voidaan jakaa kolmeen ryhmään: *toiminnalliset piirteet* (functional features), *käyttöpiirteet* (operational features) sekä *esityspiirteet* (presentation features). Toiminnalliset piirteet ovat sovellus-

ten tarjoamia palveluita. Käyttöpiirteet kuvaavat käyttäjän ja sovelluksen välistä vuorovaikutusta käyttäjän näkökulmasta. Esityspiirteet kuvaavat informaation esittämistä käyttäjille. Tunnistetut piirteet nimetään ja nimeämisen yhteydessä mahdollisesti syntyneet konfliktit ratkotaan. Piirteiden synonyymit puolestaan kirjataan aihepiirin terminologian sisältävään sanakirjaan.

Tunnistettujen piirteiden abstrahointi ja luokittelu piirremalliksi tehdään sijoittelemalla piirteet *koostumissuhteita* (consists-of) käyttäen hierarkkiseen *piirremalliin* (feature model). Piirremalli esittää aihepiirin sovellusten aihepiirikohtaiset ja sovelluskohtaiset piirteet sekä niiden väliset suhteet ja toimii siten myös käyttäjien ja kehittäjien yhteisenä kommunikointikanavana. Piirremallissa on esitettävä kaikki tunnistetut piirteet riippumatta siitä ovatko ne pakollisia, vaihtoehtoisia vai valinnaisia. Jokainen piirre on määriteltävä ja kuvauksen on myös kerrottava onko kyseessä käännösaikainen, aktivointiaikainen vai suoritusaikainen piirre.

Lopuksi on vielä vahvistettava, että tuotettu piirremalli kuvaa aihepiiriä riittävällä tarkkuudella. Tämä onnistuu aihepiiriasiantuntijoiden sekä olemassa olevien sovellusten avulla. Aihepiirianalyysin aikana käytettyjen aihepiiriasiantuntijoiden ei pitäisi osallistua tähän vaiheeseen puolueellisuuden välttämiseksi. Lisäksi pitäisi käyttää vähintään yhtä sellaista sovellusta joka ei ollut mukana aihepiirianalyysissä piirremallin soveltuvuuden ja yleispätevyyden varmistamiseksi.

Käsitemallintamisen (entity-relationship modelling) tarkoitus on tuottaa *käsitemalli* (entity-relationship model). Käsitemalliin tallennetaan sovelluskehityksessä välttämätöntä aihepiiritietoa. Aihepiiritieto esitetään aihepiirin käsitteiden sekä niiden välisten suhteiden avulla. Käsitemallin pääasiallinen tarkoitus on tukea aihepiirin ongelmien analysointia ja ymmärtämistä sekä aihepiirin sovelluskehitystä. Käsitemalli sisältää kohdeaihepiirin abstraktion.

Toiminnallisen analyysin (functional analysis) aikana tunnistetaan aihepiirin sovellusten aihepiirikohtaisia sekä sovelluskohtaisia toiminnallisia vaatimuksia. Toiminnallisen analyysin lopputuloksena saadaan aihepiirin sovellusten toiminnallisuutta kuvaava *toiminnallinen malli* (functional model). Toiminnallisen mallin tuottamisessa hyödynnetään aiemmissa vaiheissa tuotettua piirremallia sekä käsitemallia. Abstrakti toiminnallinen malli luodaan pakollisten piirteiden ja käsitteiden pohjalta. Vaihtoehtoiset ja valinnaiset

piirteet lisätään mallia tarkennettaessa. Tietoa olemassa olevien sovellusten toiminnallisuudesta saadaan *uudelleenmallintamalla* (re-engineering) vaatimusdokumenteista tai *takaisinmallintamalla* (reverse engineering) suunnitteludokumenteista tai koodista. Toiminnallista mallia käytetään aihepiirikohtaisten ongelmien ymmärtämiseen sekä vaatimusmäärittelyyn.

Aihepiirianalyysin lopputuloksena saadaan *aihepiirimalli* (domain model). Aihepiirimalli käsittää edellisissä vaiheissa tuotetun käsitemallin, piirremallin sekä toiminnallisen mallin lisäksi aihepiirin sanaston.

Arkkitehtuurimallintamisen (architecture modelling) tarkoitus on tarjota ratkaisut aihepiirimallintamisen aikana kartoitettuihin ongelmiin. Arkkitehtuurimallintamisen aikana tuotettu *arkkitehtuurimalli* (architectural model) on korkean tason suunnitelma aihepiirin sovelluksista. Sitä voidaan hyödyntää esimerkiksi yksityiskohtaisessa suunnittelussa, komponenttien rakentamisessa sekä viitemallina uusien sovellusten rakentamisessa. Se on määritelty usealla eri abstraktiotasolla, joista voidaan valita tarkoitukseen sopivin. Kerrosarkkitehtuuri auttaa myös lokalisoimaan tekniikassa ja vaatimuksissa tapahtuvia muutoksia.

3.3 Suunnittelu

Mernik, Heering ja Sloane [MHS05] kuvaavat sovelluskielen suunnitteluprosessien luokittelun kahden kriteerin perusteella: sovelluskielen suhde olemassa oleviin ohjelmointikieliin sekä suunnittelun formaalius.

Helpoin tapa suunnitella sovelluskieli on rakentaa se olemassa olevan ohjelmointikielen pohjalta. Sovelluskielen omaksuminen on myös helpompaa pohjana käytettyä kieltä aiemmin käyttäneille ohjelmoijille.

Olemassa olevan ohjelmointikielen hyödyntäminen voidaan edelleen jakaa kolmeen tapaukseen. Ensimmäisessä eli *osahyödyntämisen* (piggyback) tapauksessa olemassa oleva ohjelmointikieli sisältää jo valmiiksi kaikki tarvittavat aihepiirikohtaiset notaatiot. Suunnitteluvaiheessa tarvitsee vain valita mitkä niistä otetaan mukaan sovelluskieleen. Myös toisessa eli *erikoistamisen* (specialization) tapauksessa olemassa oleva ohjelmointikieli sisältää valmiiksi kaikki tarvittavat aihepiirikohtaiset notaatiot, mutta tässä tapauksessa siitä kuitenkin rajataan pois muut kuin aihepiirikohtaiset notaatiot. Näitä kahta

tapaa käytetään yleensä silloin kun notaatiot ovat yleisesti tunnettuja ja ne esiintyvät olemassa olevissa ohjelmointikielissä. Kolmannessa, eli *laajentamisen* (extending) tapauksessa mikään olemassa oleva ohjelmointikieli ei sisällä kaikkia tarvittavia aihepiirikohtaisia notaatioita. Tällöin ainoaksi vaihtoehdoksi jää puuttuvien aihepiirikohtaisten notaatioiden lisääminen johonkin olemassa olevaan ohjelmointikieleen. Pohjana käytetyn ohjelmointikielen kaikki notaatiot jäävät usein osaksi uutta kieltä. Tässä tapauksessa haasteeksi voi muodostua aihepiirikohtaisten notaatioiden saumaton integrointi pohjana käytetyn ohjelmointikielen notaatioiden kanssa.

Jos olemassa olevia ohjelmointikieliä ei voida tai haluta hyödyntää, sovelluskieli on luotava alusta alkaen. Uuden sovelluskielen luominen puhtaalta pöydältä voi kuitenkin olla erittäin haastavaa. Osa yleisesti tunnetuista yleisen tason ohjelmointikielten suunnittelukriteereistä pätee sovelluskielillekin. Näitä ovat esimerkiksi luettavuus ja yksinkertaisuus. Sovelluskielen suunnittelijan on kuitenkin otettava huomioon sovelluskielen erityispiirteet. Lisäksi sovelluskielen käyttäjät eivät välttämättä ole ohjelmoijia. Suunnittelijan ei myöskään pitäisi pyrkiä parantamaan aihepiirin vakiintuneita notaatioita vaan ottaa ne käyttöön sellaisenaan.

Varsinainen suunnittelu voidaan suorittaa joko vapaamuotoisesti tai formaalisti. Vapaamuotoisessa suunnittelussa sovelluskielen määrittely annetaan yleensä luonnollisella kielellä ja se saattaa sisältää joukon sovelluskielellä kirjoitettuja esimerkkisovelluksia. Formaalissa suunnittelussa sovelluskielen määrittely tehdään jollakin semantiikan määrittelykeinolla. Laajimmin käytetyt formaalit notaatiot ovat syntaksin määrittelyyn käytetyt säännölliset ilmaukset (regular expressions) ja kieliopit, attribuuttikieliopit, uudelleenkirjoitusjärjestelmät (rewrite systems) sekä semanttisiin määrittelyihin käytetyt abstraktit tilakoneet.

Vapaamuotoinen lähestymistapa koetaan usein helpommaksi. Formaalia lähestymistapaa ei kuitenkaan kannata unohtaa, koska formaalin määritelmän kehittäminen sekä syntaksista että semantiikasta voi paljastaa niissä piileviä ongelmia jo suunnitteluvaiheessa. Lisäksi toteutustyömäärää voidaan vähentää tuottamalla formaalista suunnitelmasta automaattisesti toteutus ohjelmointikielenkehitysjärjestelmien ja työkalujen avulla. Vapaamuotoista suunnittelua käytetään yleensä olemassa olevaa kieltä hyödynnettäessä kun taas formaalia suunnittelua käytetään enemmän uuden sovelluskielen suunnittelun yhteydessä.

3.4 Toteutus

Sovelluskielen suunnittelun jälkeen on valittava sopivin toteutustapa. Monet sovelluskielten toteutustavoista kuitenkin poikkeavat yleisen tason ohjelmointikielten vastaavista. Mernik, Heering ja Sloane [MHS05] listaavat kuusi sovelluskielen toteutustapaa: *tulkki* (interpreter), *kääntäjä* (compiler), *esikäsittelijä* (preprocessor), *sulauttaminen* (embedding), *laajennettava tulkki tai kääntäjä* (extensible compiler or interpreter) sekä *valmiskomponentit* (Commercial Off-The-Shelf).

Tulkin tapauksessa sovelluskielen rakenteet tunnistetaan ja tulkitaan standardin nouda, pura, suorita -syklin mukaisesti. Tämä lähestymistapa sopii dynaamisille sovelluskielille sekä silloin kun suoritusnopeus ei ole ongelma. Tulkitsemisen edut kääntämiseen nähden ovat yksinkertaisuus, parempi suoritusympäristön kontrolli sekä helpompi laajennettavuus. Kääntäjän ollessa kyseessä sovelluskielen rakenteet käännetään peruskielen rakenteiksi ja kirjastokutsuiksi. Sovelluskielellä kirjoitetulle määrittelylle tai ohjelmalle voidaan tehdä täydellinen staattinen analyysi. Sovelluskielten kääntäjiä kutsutaan usein sovellusgeneraattoreiksi. Tulkkeja ja kääntäjiä hyödyntävillä lähestymistavoilla on useita etuja. Sovelluskielen syntaksi on lähellä aihepiirin asiantuntijoiden käyttämää merkintätapaa. Hyvä virheraportointi on mahdollista. Aihepiirikohtainen analyysi, verifiointi, optimointi, rinnakkaistaminen sekä muuntaminen on mahdollista. Näillä lähestymistavoilla on myös haittapuolia. Toteuttaminen on työlästä, sillä monimutkainen sovelluskielenkäsittelijä joudutaan toteuttamaan. Sovelluskieli joudutaan todennäköisesti toteuttamaan puhtaalta pöydältä. Ohjelmointikielen laajentaminen on vaikeaa, koska useimpia ohjelmointikielen käsittelijöitä ei ole suunniteltu laajentamista silmällä pitäen. Yllä lueteltuja haittoja voidaan kuitenkin vähentää automatisoimalla ohjelmointikielenkäsittelijän toteutus ohjelmointikielenkehitysjärjestelmän tai työkalujen avulla. Tämä kuitenkin edellyttää formaalia suunnittelua ja toteutusta.

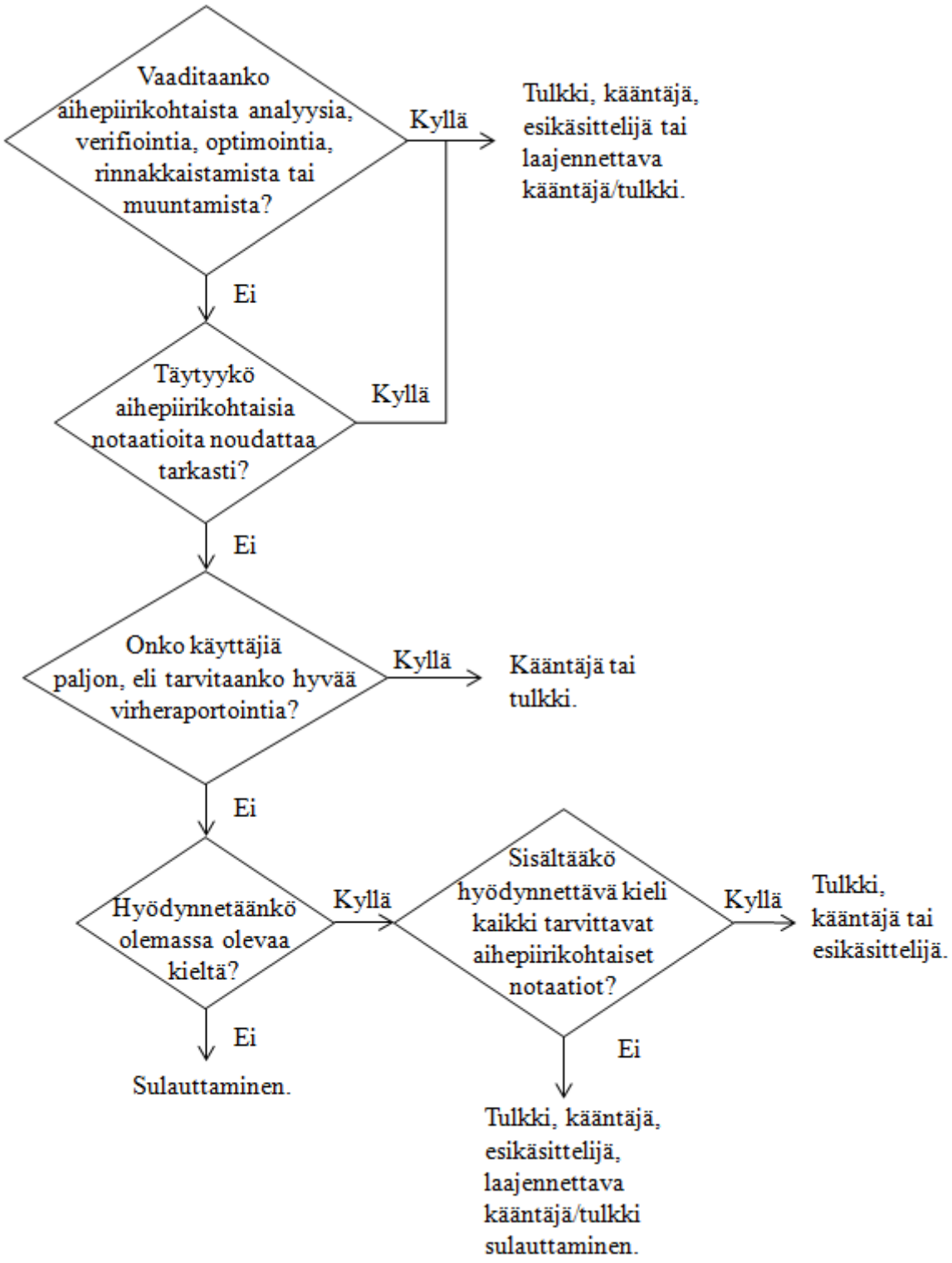
Esikäsittelijää käytettäessä sovelluskielen rakenteet käännetään peruskielen rakenteiksi. Staattinen analyysi rajoittuu peruskielen käsittelijän tarjoamaan. Esikäsittelymenetelmiä on neljä. *Makrokäsittelyn* (macro processing) avulla voidaan laajentaa makromäärittelyjä. Makrolaajennukset ovat usein riippumattomia peruskielen syntaksista. *Lähdekoodista lähdekoodiin muunnoksen* (source-to-source transformation) avulla sovelluskielellä kirjoitettu ohjelma voidaan kääntää peruskielelle. *Putkessa* (pipeline) usea sovelluskielen käsittelijä käsittelee sovelluskielen osakielellä kirjoitettuja ohjelmia. Tyypillisesti

sovelluskielen käsittelijät ovat sarjassa ja yhden käsittelijän tuottama koodi toimii seuraavan käsittelijän syötteenä. *Leksikaalisessa prosessoinnissa* (lexical processing) suoritetaan vain yksinkertainen leksikaalinen skannaus ilman syntaksianalyysia.

Sulauttamisessa olemassa olevaa yleisen tason ohjelmointikieltä laajennetaan määrittelemällä sovelluskielen rakenteet toteuttavia uusia abstrakteja datatyyppejä ja operaattoreita. Tässä tapauksessa sovelluskieli käsittää siis pohjana käytetyn ohjelmointikielen notaatioiden lisäksi aihepiirikohtaiset notaatiot. Pohjana käytetty ohjelmointikieli voi olla mikä tahansa, mutta erityisesti funktionaaliset ohjelmointikielet kuten Haskell ovat olleet suosittuja valintoja. Sulauttaminen toteutetaan yleensä sovelluskirjastojen avulla. Sulauttamisella on monia etuja. Toteutus ei vaadi paljoakaan työtä koska olemassa olevaa toteutusta voidaan uusiokäyttää. Lopputuloksena saadaan usein muihin menetelmiin verrattuna voimakkaampi sovelluskieli, koska useat notaatiot tulevat ilmaiseksi peruskielen mukana. Kohdekielen infrastruktuuri eli sen kehitystyökalut voidaan uusiokäyttää. Koulutuskustannukset voivat jäädä alhaisemmiksi koska osa käyttäjistä voi tuntea pohjana käytetyn ohjelmointikielen. Näillä lähestymistavoilla on myös haittapuolia. Syntaksi on usein kaukana optimaalisesta sillä useimmat ohjelmointikielet eivät salli mielivaltaisia syntaksin laajennuksia. Olemassa olevien operaattoreiden kuormittaminen voi aiheuttaa hämmennystä jos uusi semantiikka eroaa vanhasta. Huono virheraportointi sillä virheilmoitukset ovat pohjakielen käsitteillä aihepiirin käsitteiden asemesta. Aihepiirikohtaisten optimointien ja muunnosten saavuttaminen voi olla hankalaa, joten tehokkuus voi kärsiä erityisesti funktionaalisia ohjelmointikieliä käytettäessä. Tosin näitä haittoja voidaan minimoida. Haskelin yhteydessä voidaan käyttää monadeja toteutuksen modularisointiin. Aihepiirikohtaiseen optimointiin voidaan käyttää käyttäjän määrittelemiä muunnossääntöjä kääntäjälle.

Laajennettavan kääntäjän tai tulkin tapauksessa yleisen tason ohjelmointikielen kääntäjää tai tulkkia laajennetaan aihepiirikohtaisilla optimointi säännöillä ja koodin generoinnilla. Tulkit ovat yleensä kohtuullisen helppoja laajentaa. Kääntäjät sitä vastoin eivät, ellei niitä ole erityisesti suunniteltu laajennettaviksi.

Valmiskomponentteja käytettäessä aihepiirikohtaiset notaatiot koostetaan olemassa olevia työkaluista sekä notaatioista rajaamalla niitä aihepiirin sääntöjen mukaisesti. Mikään ei myöskään estä yhdistelemästä edellä lueteltuja menetelmiä parhaan mahdollisen lopputuloksen saavuttamiseksi.



Kuva 3. Sovelluskielen toteutustavan päätöskaavio. Mukailtu lähteestä [MHS05].

Kuva 3 esittää päätöskaavion toteutustavan valintaan. Jos sovelluskieli suunnitellaan puhtaalta pöydältä eikä olemassa olevista ohjelmointikielistä löydy yhdenmukaisuuksia, se kannattaa toteuttaa sulauttamalla. Poikkeuksen tähän muodostavat tilanteet joissa edellytetään aihepiirikohtaista analyysia, verifiointia, optimointia, rinnakkaistamista, muuntamista tai aihepiirikohtaisia notaatioita on tarkasti noudatettava tai käyttäjiä on

paljon.

Jos sovelluskieli sisältää osan olemassa olevasta ohjelmointikielestä, kannattaa myös sen toteutus uusiokäyttää. Tämän lisäksi toteutustavan valinta riippuu siitä miten olemassa olevaa kieltä aiotaan hyödyntää. Jos se sisältää kaikki tarvittavat notaatiot kuten osahyödyntämisen ja erikoistamisen tapauksissa, toteutus voidaan tehdä tulkin, kääntäjän tai esikäsittelijän avulla. Laajentamisen tapauksessa olemassa oleva ohjelmointikieli sisältää vain osan aihepiirikohtaisista notaatioista ja siihen on tarkoitus lisätä aihepiirikohtaisia notaatioita. Toteutusvaihtoehtojen kirjo on laajempi ja käsittää tulkin, kääntäjän, esikäsittelijän, laajennettavan kääntäjän tai tulkin ja sulauttamisen.

Toteutustyömäärä	**Käyttäjäystävällisyys**
Sulauttaminen	Kääntäjä
Laajennettava kääntäjä tai tulkki	Tulkki
Esikäsittelijä	Laajennettava kääntäjä tai tulkki
Tulkki	Sulauttaminen
Kääntäjä	Esikäsittelijä

Taulukko 1. Toteutustapojen vaatima työmäärä ja saavutettava hyöty laskevassa järjestyksessä.

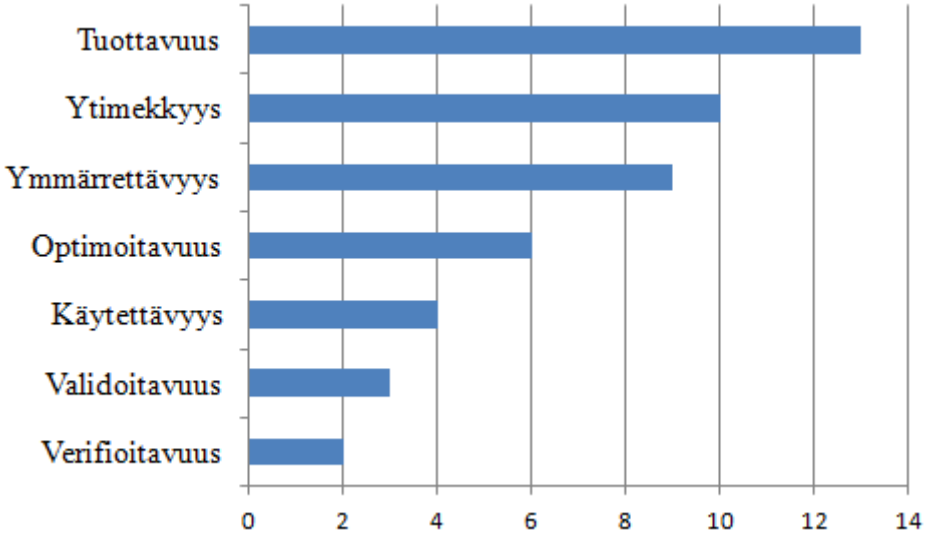
Taulukko 1 esittää eri toteutustapojen vahvuuksia järjestysasteikolla suhteessa toisiinsa. Kääntäjä on selvästi työläin. Tämän jälkeen tulevat työmäärään nähden laskevassa järjestyksessä tulkki, esikäsittelijä, laajennettava kääntäjä tai tulkki sekä sulauttaminen. Sen sijaan käyttäjäystävällisyyttä tarkasteltaessa kääntäjä ja tulkki edustavat parhaimmistoa. Niiden jälkeen tulevat laskevassa järjestyksessä laajennettava kääntäjä tai tulkki, sulauttaminen sekä esikäsittelijä. Useasta vaihtoehtoisesta toteutustavasta kannattaa valita se, josta koetaan saatavan eniten hyötyä työmäärään suhteutettuna. Käytännössä määräävänä tekijänä on kuitenkin useimmiten toteuttajan kokemus.

4 Sovelluskielen käytöstä seuraavat hyödyt ja haitat

Tässä kappaleessa käydään läpi keskeisimmät sovelluskielen käytöllä saavutettavista hyödyistä ja haitoista kehitystyön sekä sovelluksen laadun näkökulmista.

4.1 Vaikutukset kehitystyöhön

Kirjallisuudessa tunnistetut sovelluskielen käytöllä kehitystyössä saavutettavat hyödyt voidaan jakaa kehitystyön laatutekijöihin kuvan 4 esittämällä tavalla. Sovelluskielen käytön koetaan hyödyttävän kehitystyötä erityisesti parantuneen tuottavuuden, ytimekkyyden sekä ymmärrettävyyden kautta.



Kuva 4. Kirjallisuudessa tunnistetut kehitystyössä saavutetut hyödyt.

Tuottavuus (productivity) [KMB96] kuvaa miten paljon ohjelmointikielellä tyypillisesti saadaan aikaan tietyssä ajassa. Tuottavuutta mitataan tyypillisesti joko koodirivien tai toimintopisteiden perusteella. Tuottavuus on mainittu seuraavissa lähteissä: [MHS05] [HMS03] [DKV00] [BTS94] [Cle88] [DeK98] [KMB96] [GrK03] [KlB95] [HeB88] [SiB99] [HKN85] [Bar85].

Ytimekkyys (conciseness, expressiveness) [GrK03] kuvaa ohjelmointikielen semanttista etäisyyttä aihepiirin käsitteistä. Sovelluskieli sisältää aihepiirin semantiikan ja voidaan siten nähdä korkeimman asteen abstraktiona [Hud96]. Ytimekkyys on mainittu seuraavissa lähteissä: [MHS05] [HMS03] [Bar85] [BTS94] [DeK98] [GrK03] [HeB88] [SiB99] [HKN85] [Kru92].

Ymmärrettävyys (undestandability) [HMS03] kuvaa ohjelmointikielellä kirjoitetun koodin ymmärrettävyyttä. Ymmärrettävyys on mainittu seuraavissa lähteissä: [HMS03] [DKV00] [Bar85] [BTS94] [Cle88] [DeK98] [HeB88] [SiB99] [HKN85].

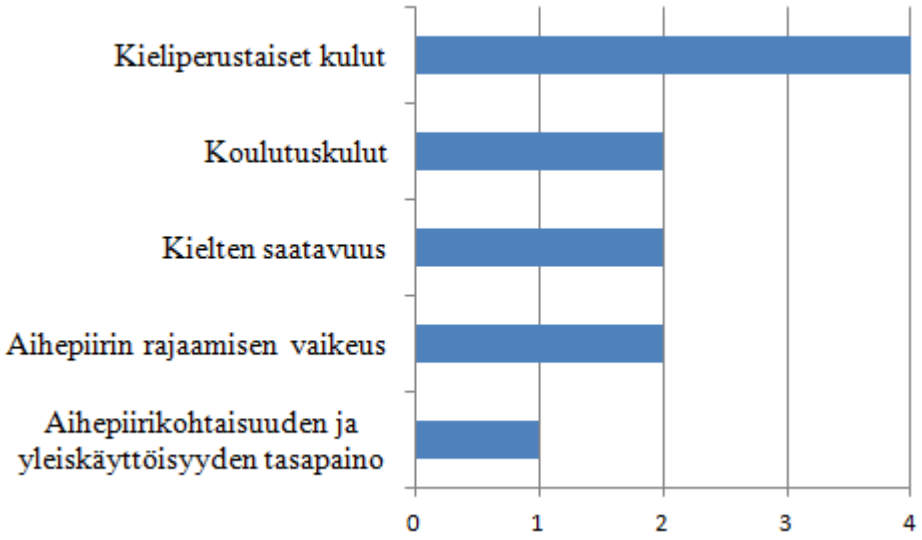
Optimoitavuus (optimizability) [MeK99] kuvaa miten helposti sovellusta voidaan optimoida toimimaan tehokkaammin tai käyttämään vähemmän resursseja. Optimoitavuus on mainittu seuraavissa lähteissä: [MHS05] [Bas97, s.29] [MeK99] [Cle88] [Nei84] [Kru92].

Käytettävyys (usability) [KMB96] kuvaa miten helppoa sovelluskielellä on kirjoittaa sovelluksia. Käytettävyys on mainittu seuraavissa lähteissä: [KMB96] [Cle88] [DeK98] [MHS05].

Validoitavuus (validibility) [DeK98] kuvaa miten helposti ohjelman oikea toiminta voidaan varmistaa. Validointi vastaa kysymykseen onko rakennettu oikea tuote eli vastaako sovellus asiakkaan tarpeita. Validoitavuus on mainittu seuraavissa lähteissä: [Bas97, s.138] [Cle88] [DeK98].

Verifioitavuus (verificability) [MHS05] [HMS03] kuvaa miten helposti ohjelmnointikielellä kirjoitettu koodi voidaan todistaa spesifikaationsa mukaiseksi. Verifiointi vastaa kysymykseen onko tuote rakennettu oikein. Verifioitavuus on mainittu seuraavassa lähteessä: [HMS03].

Kirjallisuudessa mainitut sovelluskielen käytöstä aiheutuvat haitat on esitetty kuvassa 5.



Kuva 5. Sovelluskielen käytöstä aiheutuvat haitat.

Kieliperustaiset kulut [MHS05] [DKV00] [Cle88] [Nei84] muodostuvat sovelluskielen suunnittelusta, toteutuksesta sekä ylläpidosta aiheutuvista kuluista. Sovelluskielen kehit-

täminen on vaikeaa ja vaatii sekä aihepiirin että ohjelmointikielenkehityksen asiantuntemusta [MHS05]. Toteutustavasta riippuen sovelluskielen kehittäminen saattaa tulla hyvinkin kalliiksi. Näin voi käydä esimerkiksi tilanteessa, jossa päätetään toteuttaa uusi sovelluskieli puhtaalta pöydältä kääntäjän avulla. Sovelluskieltä ja sitä kautta kääntäjää voidaan joutua muokkaamaan paljonkin jos käyttöönoton jälkeen paljastuu uusia käsittelytarpeita. Lisäksi monimutkaisen kääntäjän toteuttaminen on työlästä ja virhealtista. Suurimmat kustannukset aiheutuvat siis uuden sovelluskielen toteuttamisesta puhtaalta pöydältä kääntäjän avulla. Edullisimmaksi sitä vastoin tulee jos voidaan hyödyntää jotain olemassa olevaa kieltä ja toteuttaa uusi sovelluskieli sulauttamalla.

Koulutuskulut [DKV00] [Cle88] koostuvat sovelluskielen käyttäjien kouluttamisesta. Tällä on erityisen suuri merkitys tapauksessa, jossa tulevat ohjelmointikielen käyttäjät eivät omaa aiempaa ohjelmointikokemusta, sillä myös sovelluskielellä tapahtuva ohjelmointi edellyttää tiettyä kurinalaisuutta aihepiirikohtaisuudesta huolimatta [SmB83]. Lisäksi kohdeorganisaation koko vaikuttaa koulutuskuluihin merkittävästi, sillä laajemmassa organisaatiossa koulutettavien henkilöiden määrä voi nousta suureksikin. Suurimmat kustannukset aiheutuvat siis tilanteessa, jossa sovelluskielen käyttäjiä on paljon ja heillä ei ole aiempaa ohjelmointikokemusta. Edullisimmaksi taas muodostuu tilanne, jossa käyttäjiä on vähän ja heillä on aiempaa ohjelmointikokemusta.

Sovelluskieliä on saatavilla rajoitetusti [Kru92] [Cle88]. Huolimatta erilaisten sovelluskielten suuresta lukumäärästä [DKV00] ne keskittyvät yleensä omaan kapeaan aihepiiriinsä. Tästä johtuen sopivan valmiin sovelluskielen löytäminen on usein vaikeaa tai mahdotonta. Tällöin vaihtoehdoiksi jää yleisen tason ohjelmointikielen käyttö tai uuden sovelluskielen toteuttaminen joko jonkin olemassa olevan ohjelmointikielen pohjalta tai puhtaalta pöydältä.

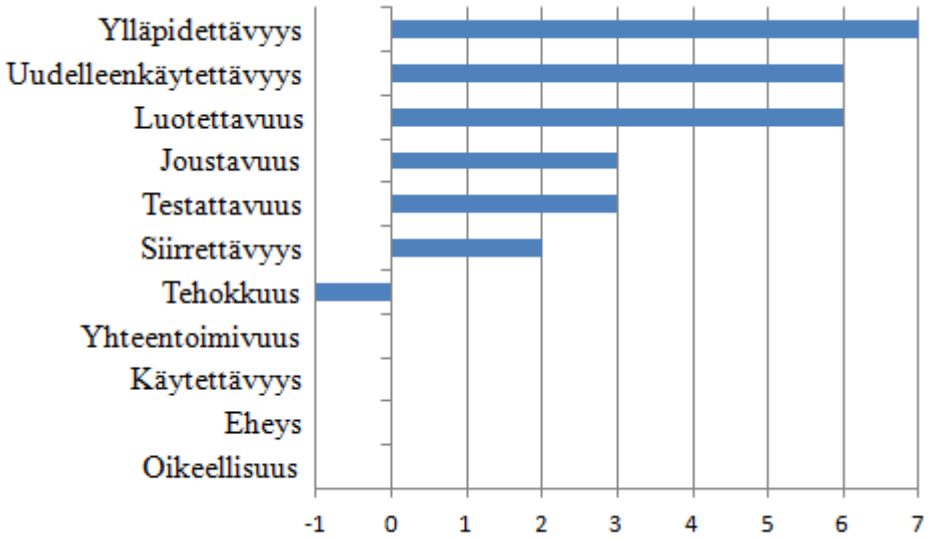
Aihepiirin rajaaminen on vaikeaa [DKV00] [Cle88]. Mitä enemmän sovelluskielen tarjoamista eduista halutaan hyötyä, sitä tarkemmin aihepiiri on rajattava. Valitettavasti rajausta tarkentamalla karsitaan samalla myös sovelluskielen käyttömahdollisuuksia. Käytännössä tämä näkyy niin että sovelluskielellä toteutettavissa olevien sovellusten määrä vähenee. Jos taas valitaan mahdollisimman laaja aihepiiri laajoja käyttömahdollisuuksia silmällä pitäen, sovelluskieli alkaa lähestyä yleisen tason ohjelmointikieltä. Tällöin menetetään sovelluskielen edut ja koko uuden sovelluskielen toteuttamisen mielekkyys kärsii. Kumpikaan edellä kuvatuista tilanteista ei ole toivottu vaan näiden kahden

ääripään väliltä tulisi löytää tilanteeseen sopiva kompromissi.

Aihepiirikohtaisten ja yleiskäyttöisten rakenteiden tasapainottaminen on haastavaa [DKV00], sillä sovelluskielen suunnittelija voi kokea sovelluskieleen sisällytettävien rakenteiden valinnan vaikeaksi. Sovelluskielen tulee kuitenkin kuvata aihepiiriään mahdollisimman tarkasti eikä aihepiirianalyysin yhteydessä kartoitettuja aihepiirin käsitteitä ole syytä mennä parantelemaan tai täydentelemään [MHS05].

4.2 Vaikutukset sovellukseen

Kirjallisuudessa tunnistetut sovelluskielen käytöllä saavutettavat hyödyt voidaan jakaa McCallin, Richardin ja Waltersin [MRW77] esittelemiin yhteentoista sovelluksen laatutekijään kuvan 6 mukaisesti. Sovelluskielen käytön koetaan hyödyttävän sovellusta eniten ylläpidettävyyden, uudelleenkäytettävyyden sekä luotettavuuden saroilla.



Kuva 6. Kirjallisuudessa tunnistetut vaikutukset ohjelmiston laatutekijöihin.

Ylläpidettävyys (maintainability) kuvaa sovelluksen ylläpidon helppoutta. Ylläpidettävyys mainitaan seuraavissa lähteissä: [MHS05] [Cle88] [DeK98] [GrK03] [HeB88] [HKN85] [Bar85].

Uudelleenkäytettävyys (reusability) kuvaa missä määrin sovellukseen liittyvää tietoa voidaan uudelleen käyttää muissa sovelluksissa. Uudelleen käytettäviä osia ovat rakenne, koodi sekä aihepiirianalyysi [MHS05]. Sovelluskielen kääntäjä tai tulkki tukee rakenteen uusiokäyttöä, sillä kaikilla sen avulla tuotetuilla sovelluksilla on sama rakenne.

Kirjastojen avulla toteutettu sovelluskieli tukee koodin uusiokäyttöä kirjastoresurssien osalta. Sovelluskieli itsessään sisältää aihepiirin semantiikan ja siten tukee aihepiirianalyysin uudelleenkäyttöä. Uudelleenkäytettävyys mainitaan seuraavissa lähteissä: [HMS03] [DKV00] [Cle88] [DeK98] [Nei84] [Kru92].

Luotettavuus (reliability) kuvaa sovelluksen tehtävistään suoriutumisen todennäköisyyttä [HMS03] [DKV00] [DeK98] [KMB96] [Cle88] [HeB88].

Joustavuus (flexibility) kuvaa miten helppoa sovellusta on muokata vastaamaan muuttuneita vaatimuksia [KMB96] [GrK03] [DeK98].

Testattavuus (testability) kuvaa sovelluksen testauksen helppoutta [SiB99] [HeB88] [HKN85].

Siirrettävyys (portability) kuvaa sovelluksen uuteen ympäristöön siirtämisen helppoutta [DeK98] [HeB88].

Tehokkuus (efficiency) kuvaa vasteaikaa tai kuormaa millä sovellus tarjoaa palveluitaan. Tästä on kirjallisuudessa ristiriitaista tietoa, sillä Batory ja kumppanit [BTS94] raportoivat tehokkuuden parantuneen sovelluskielen käytön myötä, kun sitä vastoin Deursen ja kumppanit [DKV00] sekä Mernik ja kumppanit [MHS05] laskevat sovelluskielellä kirjoitetun ohjelman potentiaalisen tehokkuushäviön erääksi sen käytöstä aiheutuvaksi haittapuoleksi. Sovelluskielen toteutustavan valinta vaikuttaa suoraan sillä kirjoitettujen ohjelmien tehokkuuteen. Erityisesti tulkin avulla toteutetuilla sovelluskielillä tehokkuus voi muodostua ongelmaksi.

Yhteentoimivuus (interoperability) kuvaa miten hyvin sovellus toimii laitteiston ja muiden sovellusten kanssa.

Käytettävyys (usability) kuvaa sovelluksen helppokäyttöisyyttä.

Eheys (integrity) kuvaa sovelluksen selviytymistä vihamielisistä toiminnoista.

Oikeellisuus (correctness) kuvaa sovelluksen yhdenmukaisuutta spesifikaationsa kanssa.

4.3 Kvantitatiivinen tutkimus

Osa edellä mainituista havainnoista on peräisin kvantitatiivisesta tutkimuksesta. Seuraa-

vassa syvennytään niiden tuloksiin.

Herndon ja Berzins [HeB88] tutkivat Kodiyak *ohjelmointikielen prototyypityskielen* käyttöä (language prototyping language) kääntäjän toteuttamisessa. Artikkelissa käydään läpi Kodiyakin avulla kehitettyjen kääntäjien kehitystyöstä saatuja kokemuksia. Kääntäjien siirrettävyys ja luotettavuus oli hyvä. Ylläpidettävyys oli erittäin hyvä ja kehitysaika lyhyt. Lisäksi kääntäjät olivat kehittyneempiä sekä useamman evoluutiokierroksen läpikäyneitä kuin perinteisellä tekniikalla toteutetut verrokit. Erityisesti Kodiyakin tarjoamat abstraktiot säästivät ohjelmoijaa puuduttavalta, uuden ohjelmointikielen semantiikkaan liittymättömältä toteutustekniseltä työltä. Yhteenvetona todetaan kokemattomien käyttäjien hyötyvän sovellusten yhtenäisistä, tarkoitukseen sopivista ja luotettavista käyttöliittymistä sekä sovellusteollisuuden hyötyvän lyhentyneestä kehitysajasta ja laskeneista kehityskustannuksista sekä laskeneista ylläpitokustannuksista.

Batory, Thomas ja Sirkin [BTS94] uudelleentoteuttivat tuotantokäytössä olevan LEAPS-kääntäjän skaalautuvan P2-kääntäjän avulla. Skaalautuvat kirjastot koostuvat primitiivisistä rakennuspalikoista sekä niitä yhdistelevästä kääntäjästä. Tarjoamiensa puhtaiden, kompaktien, korkean tason abstraktioiden ansiosta P2 mahdollisti aiemmin vaikeaselkoisina pidettyjen LEAPS-algoritmien ytimekkään esittämisen. Tästä seurasi useita etuja. Kehitystyön vaikeustaso laski huomattavasti. Kehitysaika väheni kolmannekseen. Koodirivien määrä väheni neljännekseen. Tuottavuus parani huomattavasti sekä kehitysajassa että koodiriveissä mitattuna. Suorituskyky parani 50 prosenttia. Lisäksi toimiva prototyyppi saatiin nopeasti, toteutuksen optimointi oli helppoa sekä ylläpito helpottui huomattavasti. Yhteenvedossa todetaan sovelluskielen käytön mahdollistaneen kirjoittajien ohjelmoida kuin aihepiirin asiantuntijat.

Kieburtz, McKinney ja Bell [KMB96] tutkivat sovelluskielen käyttöä viestin tulkintaan ja validointiin ilmavoimia varten kehitetyssä C^3I (command, control, communications and information) sovelluksessa. Kokeessa neljä ulkopuolisen yrityksen työntekijää suoritti samat ohjelmointitehtävät sekä sovelluskielen että aihepiiriä varten kehitettyjen *pohjien* (templates) avulla. Yhteenvedossa todetaan tuottavuuden olleen 2.92-kertainen ja luotettavuuden 2.25-kertainen aihepiirikohtaisten pohjien käyttöön verrattuna. Lisäksi sovelluskieli osoittautui joustavammaksi ratkaisuksi mahdollistaen useampien viestispesifikaatioiden käsittelyn.

Gray ja Karsai [GrK03] tutkivat kolmen sovelluskielen käyttöä kahdessa mallinnusprojektissa. Yhteenvedossa todetaan sovelluskielestä generoidun C++ koodin määrän olleen 1.49-24.62 -kertainen sovelluskielen koodimäärään verrattuna. Tuloksen hyödyllisyyttä tosin rajoittaa se, että vertailussa käytettiin sovellusgeneraattorin tuottamaa koodia kehittäjän kirjoittaman asemesta.

Klepper ja Bock [KlB95] tutkivat aihepiirikohtaisten menetelmien käyttöä osana McDonnell Douglas Aerospace Information Services Company -ilmailualan yrityksen tietojärjestelmän suunnittelua ja toteutusta. Tarkastelun kohteena olevassa järjestelmässä käytettiin sekä yleisen tason ohjelmointikieltä että sovelluskieltä siten, että osa moduuleista toteutettiin pelkästään yleisen tason ohjelmointikielellä, osassa käytettiin sekä yleisen tason ohjelmointikieltä että sovelluskieltä ja osassa käytettiin pelkästään sovelluskieltä. Lopputuloksena todetaan sovelluskielen käytön vähentäneen toteutusaikaa 25 prosentilla.

4.4 Yhteenveto sovelluskielen käytön hyödyistä ja haitoista

Kuten edellä olevista tuloksista käy ilmi, sovelluskielten käytön koetaan hyödyttävän kehitystyötä erityisesti parantuneen tuottavuuden, ytimekkyyden sekä ymmärrettävyyden kautta. Vaikuttaakin loogiselta ajatukselta, että ytimekkään ja ymmärrettävän sovelluskielen käytöstä seuraa parantunut tuottavuus.

Lopputuloksena saatava sovellus puolestaan hyötyy sovelluskielen käytöstä eniten ylläpidettävyyden, uudelleenkäytettävyyden sekä luotettavuuden saroilla. Sovelluskielen ytimekkyys ja ymmärrettävyys tulevat myös tässä esille, sillä selkeät ohjelmat ovat loogisesti ajateltuna helpompia ylläpitää sekä uudelleen käyttää. Lisäksi niissä on vähemmän virheitä.

Yllä mainitut edut eivät kuitenkaan tule ilmaiseksi. Ylivoimaisesti suurimmaksi ongelmaksi koetaan sovelluskielen suunnittelusta, toteuttamisesta sekä ylläpidosta aiheutuvat kustannukset. Tämä yhdistettynä koulutuskustannuksiin saattaa usein muodostua esteeksi uuden sovelluskielen toteuttamiselle. Lisäksi sovelluskielestä saatavasta hyödystä ei ole täyttä varmuutta ennen kuin sitä päästään käytännössä kokeilemaan. Myös aihepiirikohtaisuus aiheuttaa omat ongelmansa sovelluskielen kehitystyölle sillä aihepiirin rajaaminen on vaikeaa yhtälailla kuin tasapainon löytäminen aihepiirikohtaisten ja yleisen tason rakenteiden välillä.

Valmiiden sovelluskielten käyttöä puolestaan hankaloittaa niiden heikko saatavuus. Sovelluskielten suuresta lukumäärästä huolimatta niitä voidaan hyödyntää vain hyvin kapeilla aihepiireillä.

5 (Tietokonepeleissä käytettävät sovelluskielet)

6 (Tarkasteltavan sovelluskielen tarkempi esittely)

7 (Sovelluskielen hyödyntäminen peliohjelmoinnissa)

8 (Yhteenveto)

Lähteet

AGD02 Almeida Falbo, R., Guizzardi, G., Duarte, K. C., An ontological approach to domain engineering. Proceedings of the 14th international conference on Software engineering and knowledge engineering (SEKE'02), 15-19.07.2002, Ischia, Italia, sivut 351-358.

Bar85 Barstow, D. R., Domain-Specific Automatic Programming. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING. *IEEE Transactions on Software Engineering (TSE)*, 11, 11, 1985 (marraskuu), sivut 1321-1336.

Bas97 Basu, A., A Language Based Approach to Protocol Construction. Tohtorinväitöskirja, New York, Yhdysvallat, Cornell University, 1997.

BeJ94 Beck, K., Johnson, R., Patterns Generate Architectures. Proceedings of the 8th European Conference on Object-Oriented Programming (ECOOP'94), 4-8.7.1994, Bologna, Italia, sivut 139-149.

Ben86 Bentley, J., Programming Pearls: Little languages. *Communications of the ACM (CACM)*, 29, 8, 1986 (elokuu), sivut 711-721.

BTS94 Batory, D., Thomas, J., Sirkin, M., Reengineering a Complex Application Using a Scalable Data Structure Compiler. Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT'94), 6-9.12.1994, New Orleans, Yhdysvallat, sivut 111-120.

Cle88 Cleveland, C., Building application generators. *IEEE Software*, 5, 4, 1988 (heinäkuu), sivut 25-33.

CHW98 Coplien, J., Hoffman, D., Weiss, D., Commonality and Variability in Software Engineering. *IEEE Software*, 15, 6, 1998 (marraskuu-joulukuu), sivut 37-45.

DKV00 Deursen, A., Klint, P., Visser, J., Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35, 6, 2000 (kesäkuu), sivut 26-36.

FaS97 Fayad, M., Schmidt, D. C., Object-oriented application frameworks. *Com-*

munications of the ACM (CACM), 40, 10, 1997 (lokakuu), sivut 32-38.

FHL97 Froehlich, G., Hoover, H. J., Liu, L., Sorenson, P., Hooking into Object-Oriented Application Frameworks. Proceedings of the 19th International Conference on Software Engineering (ICSE'97), 17-23.5.1997, Boston, Yhdysvallat, sivut 491-501.

FPF98 Frakes, W., Prieto-Diaz, R., Fox, C., DARE: Domain analysis and reuse environment. *Annals of Software Engineering*, 5, 1, 1998, sivut 125-141.

GJS05 Gosling, J., Joy, B., Steele, G., Bracha, G., The Java™ Language Specification Third Edition, Addison-Wesley, 2005.

GrK03 Gray, J., Karsai, G., An Examination of DSLs for Concisely Representing Model Traversals and Transformations. Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS-36 2003), 6-9.1.2003, Big Island, Yhdysvallat, sivut 325-324.

Har04 Harrison, W., The dangers of end-user programming. *IEEE Software*, 21, 4, 2004 (heinäkuu-elokuu), sivut 5-7.

HeB88 Herndon, R. M., Berzins, V. A., The Realizable Benefits of a Language Prototyping Language. *IEEE Transactions on Software Engineering (TSE)*, 14, 6, 1988 (kesäkuu), sivut 803-809.

HKN85 Horowitz, E., Kemper, A., Narasimhan, B., A Survey of Application Generators. *IEEE Software*, 2, 1, 1985 (tammikuu), sivut 40-54.

Hud96 Hudak, P., Building Domain-Specific Embedded Languages. *ACM Computing Surveys (CSUR)*, 28, 4es, 1996 (kesäkuu), sivut 196-201.

Hud98 Hudak, P., Modular domain specific languages and tools. Proceedings of the Fifth International Conference on Software Reuse (ICSR'98), 2-5.6.1998, Victoria, Kanada, sivut 134-142.

IWA91 Iscoe, N., Williams, G. B., Arango, G., Domain Modeling for Software Engineering. Proceedings of the 13th international conference on Software engineering (ICSE'91), 13-17.05.1991, Austin, Yhdysvallat, sivut 340-343.

Joh97 Johnson, R., E., Components, frameworks, patterns. Proceedings of the 1997 symposium on Software reusability (SSR'97), 17-19.05.1997, Boston, Yhdysvallat, sivut 10-17.

KCH90 Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., Peterson, A. S., Feature-Oriented Domain Analysis (FODA): Feasibility Study. Technical Report, CMU/SEI-90-TR-21, Pittsburgh, Yhdysvallat, Carnegie-Mellon University, 1990.

KlB95 Klepper, R., Bock, D., Third and fourth generation language productivity differences. *Communications of the ACM (CACM)*, 38, 9, 1995 (syyskuu), sivut 69-79.

KMB96 Kieburtz, R. B., McKinney, L., Bell, J. M, Hook, J., Kotov, A., Lewis, J., Oliva, D. P., Sheard, T., Smith, I., Walton, L., A Software Engineering Experiment in Software Component Generation. Proceedings of the 18th international conference on Software engineering (ICSE'96), 25-29.03.1996, Berlin, Saksa, sivut 542-552.

Kru92 Krueger, C. W., Software Reuse. *ACM Computing Surveys (CSUR)*, 24, 2, 1992 (kesäkuu), sivut 131-183.

LaM97 Lam, W., McDermid, J. A., A Summary of Domain Analysis Experience By Way of Heuristics. Proceedings of the 1997 symposium on Software reusability (SSR'97), 17-19.5.1997, Boston, Yhdysvallat, sivut 54-64.

McI68 McIlroy, M. D., Mass-Produced Software Components. Software Engineering: Report on a conference sponsored by the NATO Science Committee, 7-11.10.1968, Garmisch, Germany, sivut 138-150.

HMS03 Heering, J., Mernik, M., Sloane, A. M., Domain-Specific Languages. Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS-36 2003), 6-9.1.2003, Big Island, Yhdysvallat, sivut 323-323.

MHS05 Mernik, M., Heering, J., Sloane, A. M., When and How to Develop Domain-Specific Languages. *ACM Computing Surveys (CSUR)*, 37, 4, 2005 (joulukuu), sivut 316-344.

Moo61 Moore, D. P., Library loading with alternate routine selection. *Communications of the ACM (CACM)*, 4, 11, 1961 (marraskuu), sivut 496-497.

MRW77 McCall, J. A., Richards, P. K., Gene, F. W., Factors in Software Quality: Concepts and Definitions of Software Quality. Technical report, RADC-TR-77-369 volume 1 (of three), Sunnyvale, Yhdysvallat, General Electric Company, 1977.

Nei80 Neighbors, J. M., Software Construction Using Components. Tohtorin väitöskirja, Irvine, Yhdysvallat, University of California, 1980.

Nei84 Neighbors, J. M., The Draco Approach to Constructing Software from Reusable Components. *IEEE Transactions on Software Engineering (TSE)*, 10, 5, 1984 (syyskuu), sivut 564-574.

Pri90 Prieto-Díaz, R., Domain Analysis: An Introduction. *ACM SIGSOFT Software Engineering Notes*, 15, 2, 1990 (huhtikuu), sivut 47-54.

RLJ99 Raggett, D., Le Hors, A., Jacobs, I., HTML 4.01 Specification: W3C Recommendation 24 December 1999. http://www.w3.org/TR/1999/REC-html401-19991224/. [Luettu 05.02.2012]

Sch97 Schmid, H. A., Systematic Framework Design by Generalization. *Communications of the ACM (CACM)*, 40, 10, 1997 (lokakuu), sivut 48-51.

SCK96 Simos, M., Creps, D., Klingler, C., Levine, L., Allemang, D., Organization Domain Modeling (ODM) Guidebook: Version 2.0. Software Technology for Adaptable, Reliable Systems (STARS), STARS-VC-A025/001/00, Manassas, Yhdysvallat, Lockheed Martin Tactical Defense Systems, 1996.

SiB99 Sirer, E. G., Bershad, B. N., Using Production Grammars in Software Testing. Proceedings of the Second conference on Domain-specific languages (DSL'99), 3-5.10.1999, Austin, Yhdysvallat, sivut 1-13.

SmB83 Smoliar, S. W., Barstow, D., Who needs languages, and why do they need them? or no matter how high the level, it's still programming. Proceedings of the 1983 ACM SIGPLAN symposium on Programming language issues

in software systems (SIGPLAN'83), 27-29.06.1983, San Francisco, Yhdysvallat, sivut 149-157.

SMT09 Sprinkle, J., Mernik, M., Tolvanen, J., Spinellis, D., What Kinds of Nails Need a Domain-Specific Hammer? *IEEE Software*, 26, 4, 2009 (heinäkuu-elokuu), sivut 15-18.

SuM04 Sutcliffe, A., Mehandjiev, N., End-user development: tools that empower users to create their own software solutions. *Communications of the ACM (CACM)*, 47, 9, 2004 (September), sivut 31-32.

TCY93 Tracz, W., Coglianese, L., Young, P., Domain-Specific Software Architecture: Engineering Process Guidelines. Technical report, ADAGE-IBM-92-02 Version 2.0, Owego, Yhdysvallat, IBM Corporation, 1993.

TTC95 Taylor, R. N., Tracz, W., Coglianese, L., Software Development Using Domain-Specific Software Architectures. *ACM SIGSOFT Software Engineering Notes*, 20, 5, 1995 (joulukuu), sivut 27-38.

Tra94 Tracz, W., Domain-Specific Software Architecture (DSSA) Frequently Asked Questions (FAQ). *ACM SIGSOFT Software Engineering Notes*, 19, 2, 1994 (huhtikuu), sivut 52-56.

Tra95 Tracz, W., DSSA (Domain-Specific Software Architecture): Pedagogical Example. *ACM SIGSOFT Software Engineering Notes*, 20, 3, 1995 (heinäkuu), sivut 49-62.