UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Davor Čretnik

Implementacija SCA specifikacije

DIPLOMSKO DELO NA VISOKOŠOLSKEM STROKOVNEM ŠTUDIJU

Ljubljana, 2013

UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Davor Čretnik

Implementacija SCA specifikacije

DIPLOMSKO DELO NA VISOKOŠOLSKEM
STROKOVNEM ŠTUDIJU

MENTOR: dr. Saša Divjak

Ljubljana, 2013

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00402/2013
Datum: 02.04.2013

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **DAVOR ČRETNIK**

Naslov: **IMPLEMENTACIJA SCA SPECIFIKACIJE**
IMPLEMENTATION OF SCA SPECIFICATION

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija prve stopnje

Tematika naloge:

Posvetite se problematiki povezovanja več servisov v homogeno celoto v sklopu velikih heterogenih sistemov. Pri tem se naslonite na specifikacijo SCA, ki omogoča modularnost in neodvisnost od posameznih produktov. Razložite metode, mehanizme in načine konfiguriranja povezovalnega sistema. Demonstrirajte uporabo koncepta s primernim testnim okoljem.

Mentor:

prof. dr. Saša Divjak

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:

prof. dr. Nikolaj Zimic

Povzetek

Diplomsko delo se ukvarja s problematiko srednje velikih in velikih produktov oziroma sistemov, ki morajo povezovati več servisov skupaj v homogeno celoto. Takšni sistemi so problematični v smislu posodabljanja in zamenjave posameznih delov. Implementacija SCA specifikacije omogoča veliko mero modularnosti in neodvisnosti produkta.

Sicer heterogeni sistem tako lahko predstavimo kot skupek neodvisnih komponent. Vsaka komponenta, ki sestavlja celoto, je enostavno zamenljiva, kar pomeni, da ni potrebno celotnega sistema na novo prevajati s prevajalnikom.

Osrednji del SCA implementacije tvori Sca vsebovalnik, katerega delovanje je enostavno razloženo. Prav tako so prikazani možni scenariji uporabe. Za pravilno delovanje Sca vsebovalnika sta potrebna torvarna servisov in servisni register, katerih delovanje je prav tako razloženo.

V diplomskem delu so razložene metode, mehanizmi in načini konfiguracije povezovalnega sistema. Razložene so rešitve vsakdanje programerske problematike, kot so večnitnost, množičnost in povezovanje različnih implementacij različnih C++ razredov v delujoči sistem, imenovan kompozit. Kompozit teče znotraj dodeljenega konteksta, ki je definiran s skupnimi servisi, ki jih uporavljajo vse komponente.

Razloženi so posamezni elementi kompozita, ki je sestavljen v XML obliki, ter njegove lastnosti.

Demonstracija uporabe na primeru testnega okolja prikaže, kako se ogrodje obnese v resničnem scenariju. Razloženi so posamezni elementi konfiguracije, prav tako je prikazana struktura sistema na podatkovnem sistemu.

Ključne besede: SCA, kompozit, XML, servisi

Draft

The thesis is covering an area of solutions for medium to large (enterprise) products or systems, which are supposed to tie in many different services into one homogenous entity. These kinds of systems are problematic in sense of updating and replacing specific parts of them. This is where an implementation of SCA specifiacion comes to provide modularity and independence of a product on a large scale.

The otherwise heterogen systems can be presented as a set of independent components. Each of the components, which build up the whole, is easily replacable. This means that there is no need to recompile the whole system with a compiler.

Main part of the SCA implementation consists of Sca container. It's operation is explained in a simple manner, as are it's scenarios of use. In order for Sca comtainer to function properly, service factory and service registry are introduced. Their means of operation is also explained.

This thesis also consists of explanations of methods, mechanisms and different means of configuring the system. Everyday programming problems such as multithreading, multiplicity and wiring of different implementations of different C++ classes into a functional system, named composite, are also explained. A composite is being run in an assigned context, which is defined by common services that are being used by all components.

All elements of a composite, which are presented in XML form, are explained as well as the properties of it.

In the last part of the thesis, there is a demonstration of a use case in a testing environment, explaining how the developed system behaves and performs in a real life scenario. Elements of configuration are explained step by step along with a presentation of how the system structure looks like on the filesystem.

Keywords: SCA, composite, XML, services

1. Uvod

Dandanes se mnogo razvijalcev in posledično tudi podjetij srečuje z dokaj splošnim problemom združljivosti programov oziroma programske kode znotraj istega produkta. Največkrat se stremi k temu, da bi bila arhitektura programa čim bolj fleksibilna, vendar je takšna naloga zelo težavna, saj se lahko že v okviru enega programa srečamo z mnogo različnimi protokoli, nastavitvami programa in celo z različnimi programskimi jeziki. Rešitev tega problema leži v modularnosti ali, z drugimi besedami povedano, v zamenljivosti določenega dela programske kode. Tak način gradnje programa nam omogoča hitrejše, varnejše in predvsem cenejše posodabljanje.

Točno v tem pogledu nastopi SCA – Service Component Architecture oziroma arhitektura servisnih komponent. To je način gradnje programov na principu modularnosti, katerega osnovni gradnik je komponenta. Takšna rešitev nam na abstraktnem nivoju natančno opiše, kako se lahko posamezni deli programske kode sporazumevajo, kar privede do enostavnega in celostnega programiranja. Edini potrebni vodnik pri izdelavi posamezne komponente je namreč specifikacija SCA, in le-ta nam zagotavlja, da se komponente med seboj lahko povezujejo in komunicirajo.

Za osnovo arhitekture, ki sledi tem načelom, lahko vzamemo enega od mnogih obstoječih specifikacij ter ga implementiramo v poljubnem programskem jeziku. V tej diplomski nalogi se bomo ukvarjali z delno implementacijo OASIS specifikacije SCA[1].

[1] http://oasis-opencsa.org/sca

2. Glavni del

2.1. Kaj je SCA?

SCA je skupek pravil, ki opisujejo model izdelovanja aplikacij in sistemov v okviru servisno orientirane arhitekture (SOA[2]). SCA razširja in dopolnjuje vse dosedanje pristope k implementaciji servisov. Gradi na odprtih standardih, kot so na primer spletni servisi.

SCA temelji na tem, da so poslovne funkcije produkta podane kot vrsta servisov, ki so sestavljeni skupaj tako, da ustvarjajo rešitev točno določenih potreb uporabnika. Takšne sestavljene aplikacije lahko vsebujejo tako nove servise, ki so bili razviti za točno določeno aplikacijo, kot obstoječe servise, ki so že bili uporabljeni v kakšnem drugem sistemu ali aplikaciji. SCA ponuja model za sestavo, izdelavo in vezavo novih ali obstoječih servisov.

Takšna arhitektura zavzema vrsto tehnologij za servisne komponente in za metode dostopanja, ki so uporabljene za njihovo povezavo. To pomeni, da so v komponente vključeni različni programski jeziki in tudi različna programska ogrodja in okolja, ki so uporabljena z omenjenimi programskimi jeziki. V zvezi z metodami dostopanja SCA kompozicije dopuščajo uporabo mnogih komunikacijskih servisov in tehnologij, ki so v splošni uporabi, kot so na primer spletni servisi, sistemi za sporočanje in oddaljeno klicanje procedur (RPC[3]).

[2] Service-Oriented Architectrue
[3] Remote Procedure Call

2.2. Zakaj SCA?

Zaradi same zapletenosti sistema sestave servisov oziroma komponent se lahko porodi vprašanje, zakaj sploh uporabljati tako težaven, splošen sistem, če so na voljo lažje rešitve za točno določen problem.

Takšna implementacija je sicer bolj primerna za velike, tako imenovane »enterprise« produkte, ki so v neprestanem razvoju. Odgovor na zgoraj postavljeno vprašanje pa tiči v modularnosti ter zmanjševanju stroškov za vzdrževanje samega sistema.

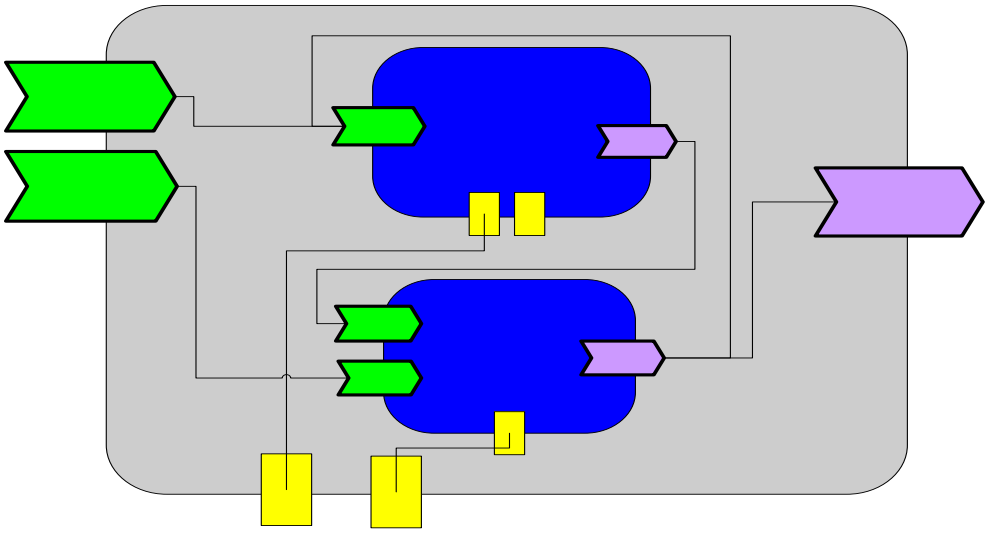
V tem primeru nam SCA pomaga pri premostitvi vsakdanjih ovir v svetu programskih jezikov, saj podpira neodvisnost binarnih, izvedljivih komponent. V praksi to pomeni, da lahko sestavimo servise, ki niso nujno prevedeni z isto verzijo prevajalnika, če je to v sklopu istega programskega jezika, ali pa so zapisani v nekem drugem programskem jeziku, vendar se podrejajo vmesnikom SCA specifikacije.

V praktičnem smislu to pomeni, da lahko imamo (v našem primeru v C/C++ programskem jeziku) določeno komponento prevedeno z različnimi prevajalniki, npr. VC90 in VC100[4], ki bo še vedno delovala znotraj celotnega sistema.

[4] Visual Studio 2008 in Visual Studio 2010 prevajalnika

2.3. Implementacija SCA vsebovalnika

Osrednja komponenta, ki skrbi za SCA povezovanje, se imenuje *vsebovalnik*, ki je neke vrste »škatla«, v katero odlagamo komponente. Za lažjo predstavo sestave SCA vsebovalnika si lahko ogledamo spodnjo ponazoritev (Slika 1). Celotno strukturo, s katero na najvišjem nivoju upravlja SCA vsebovalnik, imenujemo kompozit.



Slika 1 Skica primera kompozita v SCA vsebovalniku

Kompozit smo oblikovali tako, da je sestavljen iz komponent, ki so sestavljene iz servisov, lastnosti in referenc.

Komponente so naše implementacije C++ razredov.

Servisi so abstraktni C++ temeljni razredi (vmesniki), ki smo jih implementirali v naših komponentah.

Lastnosti so C++ razredni člani kakršnega koli tipa v naših komponentah, ki vsebujejo podatke, katere lahko vnaprej konfiguriramo v specifikaciji kompozita.

Reference so naši referenčni C++ razredni člani/kazalci na kakšen drugi C++ vmesnik (komponentni servis). Lahko so povezani z drugimi servisi v kompozitu ali pa ostanejo nepovezani, tako da jih lahko implementacija komponente dinamično poveže s pomočjo registracije servisa.

Komponentne servise, lastnosti ali reference lahko nadgradimo tudi tako, da postanejo globalni/vidni celotnemu kompozitu.

2.4. Definicija potrebnih mehanizmov za implementacijo

V nadaljnjih nekaj poglavjih so naštete in opisane lastnosti in mehanizmi, ki smo jih razvili za potrebe implementacije SCA vsebovalnika.

2.4.1. Binarna neodvisnost

Če želimo vpeljati binarno neodvisnost med komponentami, morajo biti vsi komponentni servisi (C++ abstraktni temeljni razredi) bazirani na temeljnih vmesnikih (`IntBase`). Iz tega sledi, da se vse implementacije tega vmesnika podrejajo enaki obliki virtualne tabele metod, ki so skupne vsem komponentam.

Tako izkoristimo samo obliko C++ programskega jezika za združljivost sicer nekompatibilne izvršne kode. Vmesnik zagotavlja obstoj določenih virtualnih metod, ki pa so lahko med komponentami različno implementirane, vendar se vedno enako vedejo znotraj sistema. S tem zagotovimo skladno upravljanje z vrednostmi, ki jih te metode vračajo.

Enostavneje povedano, dinamične knjižnice ali izvršne datoteke, ki so bile prevedene z različnimi prevajalniki, vendar se podrejajo temeljnim vmesnikom, lahko brez težav komunicirajo med seboj v kompozitu.

2.4.2. Temeljni vmesniki in hierarhija objektov

Temeljna hierarhija objektov je obširna tematika in ni subjekt te diplomske naloge. Na kratko bomo razložili, čemu služi in kako je zastavljena za potrebe implementacije SCA vsebovalnika.

Temeljni vmesniki so C++ vmesniki, ki definirajo najosnovnejše skupne funkcionalnosti vseh komponent. Razvili smo ga v namen poenotenja obnašanja podatkovnih tipov in komponent. Primer takšnega vmesnika je `IntBase`, kjer smo določili temeljno obliko obnašanja, kateremu se morajo vse komponente podrejati. Sem spadajo:

Operatorji:

Javni operatorji:

- Enakost (==),
- Neenakost (!=),
- Večje (>),
- Manjše (<).

Zasebni operatorji:

- Priredje (=),
- Referenca (&),
- Konstantna referenca (& const).

Metode:

- IsIdentical – preverjamo ali sta dve instanci identični
- ToString – vračamo string referenco z opisom instance komponente
- ToAdaptor – vračamo novi prirejevalnik tipa
- GetObjectRef – vračamo reference objekta
- Clone – ustvarjamo nove kopije objekta
- QueryInterface – poizvedujemo po implementaciji določenega vmesnika
- GetTypeInfo – vračamo informacije o tipu objekta

`IntBase` je glavni temeljni vmesnik, vendar le en od mnogih, ki smo jih definirali v prej omenjeni hierarhiji objektov.

2.4.3. Življenski cikel komponente

Za kontrolo življenjskega cikla komponente smo oblikovali vmesnik `IntComponent`, ki ga mora implementirati vsaka komponenta. Ta nam koristi za to, da imamo možnost kontrole življenjskega cikla, kar vključuje začetni zagon in ugašanje komponente.

Vsebino `IntComponent` servisa smo definirali tako:

Metode:

- Initialize – poženemo začetni zagon komponent v določenem kontekstu
- Uninitialize – ugasnemo komponento komponente
- Configure – konfiguriramo vsebovalnik
- IsInitalized – preverimo ali je bila komponenta primerno zagnana

Komponentam, ki nimajo implementiranih teh metod, priredimo privzete prazne bazične implementacije. S tovarno servisov[5] v tem primeru samo instanciramo implementacijo razreda komponente brez posebnega začetnega zagona. Takšen postopek je priporočljiv predvsem za vse pasivne komponente, ki ne tvorijo glavnine poslovne logike kompozita.

Za vse komponente, ki potrebujejo točno določeno začetno stanje, implementiramo začetno nastavitev članov C++ razredov v metodi `Initialize`. Pri tem je pomembno, da za vse netrivialne člane napišemo tudi kodo, ki jih pri rušenju komponente počisti za seboj v metodi `Uninitialize`.

Metoda `Configure` ni omejena samo na konfiguracijo vsebovalnika, ampak jo lahko implementiramo v vseh komponentah kompozita. Večinoma se uporablja za netrivialne konfiguracije komponent, ki niso ustrezne za uporabo pri metodi `Initialize`.

Z implementacijo `IsInitialized` pa lahko preverjamo, ali je komponenta že primerno zagnana ali ne. V prej omenjenem primeru, kjer je uporabljena privzeta prazna implementacija, je rezultat metode omejen na to, ali je bil razred komponente vzpostavljen ali ne, torej brez preverjanja morebitnih posebnih pogojev za člane C++ razredov.

[5] Tovarna servisov in njeno delovanje je opisano v poglavju 2.4.7.

2.4.4. Množičnost

Z množičnostjo, ki je lahko tipa (0..1, 1..1, 0..n, 1..n), ponudimo uporabniku zmožnost, da eno referenco lahko poveže z večjim številom servisov. Prav tako ponujamo nadaljnje možnosti dodatnega upravljanja z napakami v primeru, da se povezovanje ne ujema z definirano množičnostjo reference.

Množičnosti imajo različne pomene:

- **0..1 –** na servis lahko povežemo nobeno ali natanko eno referenco komponente,
- **1..1 –**natančno eno referenco komponente moramo obvezno povezati na ta servis,
- **0..n –** na servis lahko povežemo nobeno ali več referenc komponent hkrati
- **1..n –** na servis lahko povežemo obvezno eno ali več referenc komponent.

2.4.5. Dinamično povezovanje

Statično povezovanje je lahko omejujoče v nekaterih situacijah, kjer komponenta dinamično izbere implementacijo servisa, ki bi moral biti povezan z referenco.

V tem primeru moramo atribut SCA referenčne oznake, imenovan `'wiredByImpl'`, nastaviti na `'true'`. Tako lahko implementacijo komponente uporabimo v servisnem registru za povpraševanje po komponentah, ki so na voljo, in dinamično izberemo, kateri komponentni servis povezati.

Torej, imamo lahko več komponent, ki opravljajo isto poslovno logiko, vendar na različne načine. Želeni način delovanja sami izberemo, ali pa ga določimo z vnaprej postavljenimi pogoji.

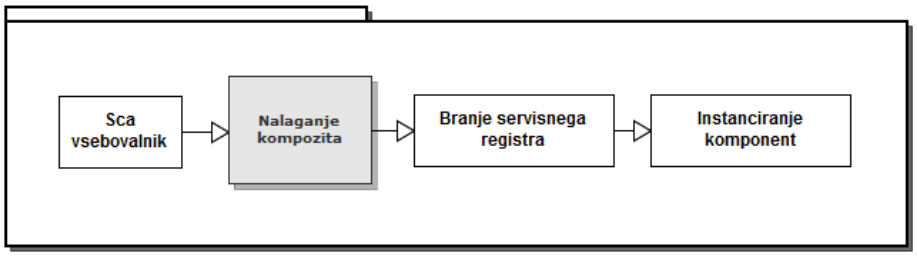
2.4.6. Servisni register (entiteta)

Servisni register potrebujemo, da se implementacijam komponent ni potrebno zavedati, kje se nahajajo komponente, ki implementirajo določen servis.

Uporabniku servisov omogočimo poizvedovanje, preko katerega ugotovi, katere komponente implementirajo določene servise, in tako izve, kje se te komponente nahajajo.

Servisni register bomo vmestili v sam sistem SCA vsebovalnika. Vsak proces, ki ga zaženemo, mora prebrati obstojen register iz datotečnega sistema.

Obstojni register smo oblikovali tako, da je sestavljen iz množice XML datotek, pri čemer vsaka XML datoteka opisuje dinamično knjižnico/komponento. V XML datoteko vpišemo informacijo o razredih, ki so implementirani, ter vmesnikih, ki jih ti razredi implementirajo.



Slika 2 Prikaz umeščenosti servisnega registra

2.4.7. Tovarna servisov

Tovarno servisov smo implementirali tako, da uporablja servisni register za ustvarjanje komponent. Uporabnik mora samo poimensko določiti, katero komponento vzpostaviti, ni mu pa potrebno vedeti, katera dinamična knjižnica implementira komponento.

Vhodne podatke za tovarno servisov smo oblikovali tako, da sestojijo iz imena vmesnika in imena komponente. S tovarno najprej preverimo v registru, ali komponenta res implementira specificirani vmesnik, ter v kateri deljeni knjižnici se komponenta nahaja. Nato uporabimo nalagalnik razredov[6], s katerim naložimo deljeno knjižnico in poiščemo abstraktno tovarno za določeno komponento.

2.4.8. Razlikovanje med različicami

V primeru, da želimo imeti vzporedni obstoj večjega števila različic iste komponente, povečamo tudi zapletenost celotne arhitekture, ker so lahko določene komponente odvisne od specifične različice katere druge komponente. S tem ustvarimo mrežo odvisnosti med inačicami komponent namesto med komponentami samimi.

Za potrebe takšnega sistema, kjer vse komponente implementirajo določene vmesnike, ki definirajo semantično obnašanje komponent, smo dolžni zagotoviti vzvratno kompatibilnost vsake nove različice komponente tako, da implementirajo iste oziroma razširjene vmesnike.

Eno različico komponente vključimo v register, tako da več različic iste komponente ne bo mogoče namestiti na posamezen sistem.

2.4.9. Upravljanje z napakami

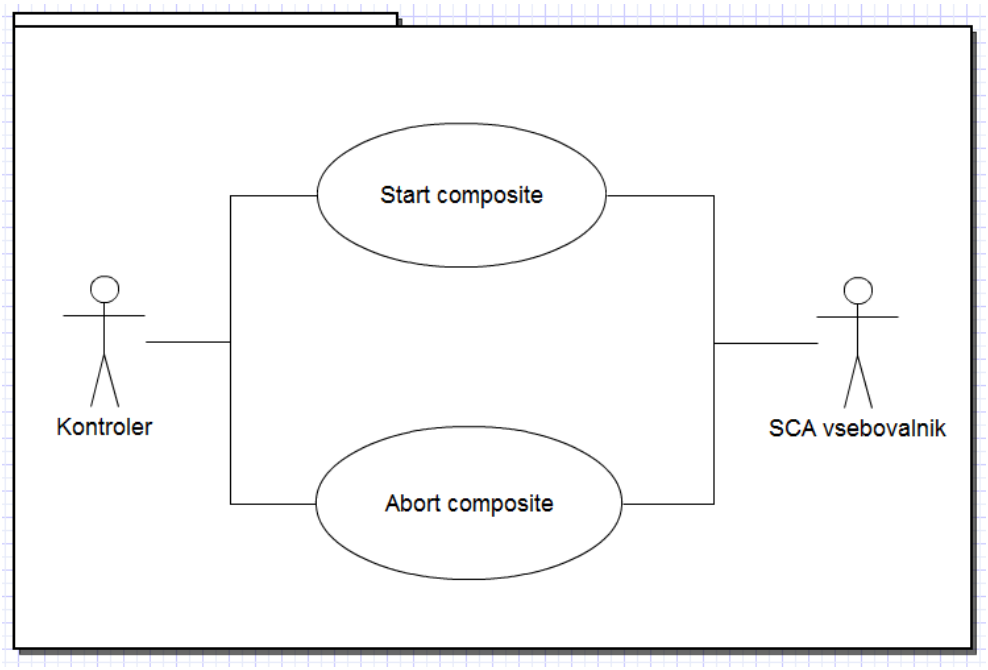
Upravljanje z napakami smo zasnovali tako, da preverja za duplikati komponent (komponente z istim imenom, ki se nahajajo v različnih deljenih knjižnicah). Ko duplikat najde, ga pri vzpostavitvi servisnega registra preskoči in zapiše dogodek v dnevniški zapis. Enako naredi v primeru ugotovitve, da je nameščenih več različic iste komponente. Za uporabo vedno izbere najvišjo različico komponente, ker se predpostavlja, da le-ta vsebuje trenutno željeno poslovno logiko.

[6] Nalagalnik razredov uporabi domorodni vmesnik API za nalaganje knjižnic

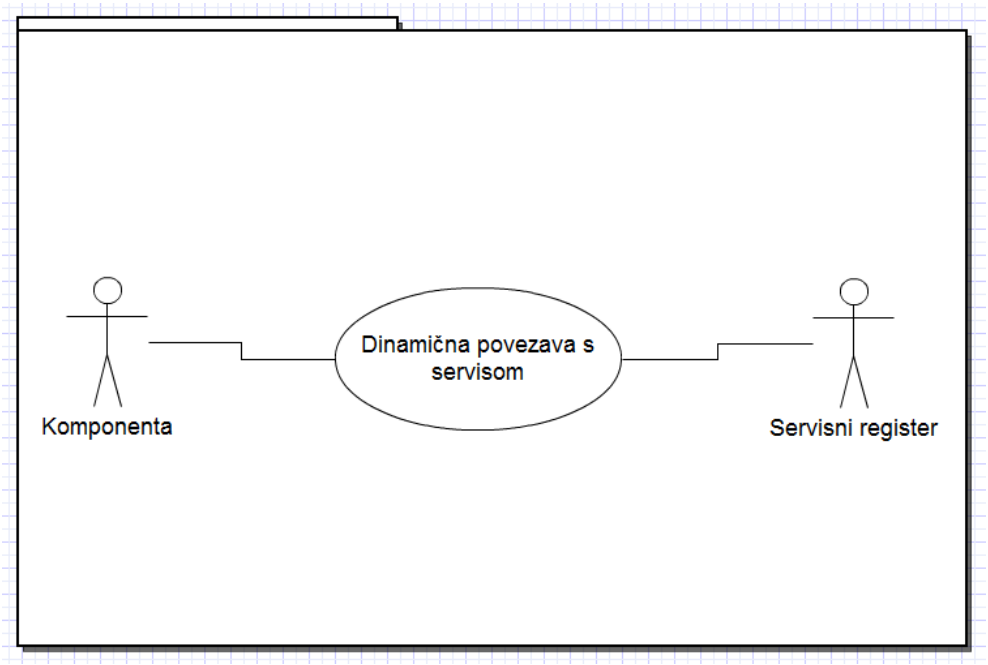
2.5. Obnašanje SCA vsebovalnika

2.5.1. Primeri uporabe

Za lažjo predstavo smo glavna primera uporabe upodobili na slikah Slika 3 in Slika 4.



Slika 3 Model primera uporabe kompozita



Slika 4 Model primera uporabe dinamičnih povezav

2.5.2. Scenariji uporabe

S to implementacijo SCA lahko izvedemo enega izmed mnogih scenarijev, ki vključujejo:

- **Start composite –** Kontroler uporabi SCA vsebovalnik za nalaganje in zaganjanje kompozita.
- **Abort composite –** Kontroler registrira kompozit v sistemu za notifikacije. Ko je Abort signaliziran, se vse niti v kompozitu prekinejo. Implementacija komponente je odgovorna za periodično preverjanje, ali je bil abort signal sprejet, in čiščenje za seboj.
- **Dinamično povezovanje –** Komponenta v servisnem registru poišče prost servis in se dinamično poveže.

2.6. Mehanizmi

Za učinkovito delovanje celotne arhitekture potrebujemo posebne mehanizme, ki jih bomo opisali v sledečih podpoglavjih. Komponente v kompozitu morajo biti povezane v (globalni) kontekst, saj lahko le tako komunicirajo med seboj. V namen takšnega povezovanja je implementiran servisni register. Pri tem moramo upoštevati tudi večnitnost sistema, tipizacijo lastnosti, tipizacijo referenc in množičnost.

2.6.1. Ravnanje s kontekstom

Kontekst omogoča vsem komponentam dostop do skupnih storitev, ki so del okolja, v katerem tečejo:

- Logger
- Service factory
- Environment handler

Te servise smo implementirali enako kot komponente, ki so naložene in konfigurirane skozi kontekst. Omenjene komponente bi lahko bile konfigurirane tudi skozi specifikacijo kompozita, vendar glede na to, da vse komponente v kompozitu potrebujejo iste servise, bi hitro postalo nadležno in nagnjeno k napakam definirati vedno iste reference in povezave do skupnih servisov za vsako komponento posebej.

V ta namen smo ustvarili ločeno konfiguracijo glavnega konteksta, ki je v XML obliki. Primer take XML konfiguracije:

```
<Context>
	<ServiceFactory>
		<Implementation>PATH[7]\ServiceFactory.dll</Implementation>
		<Configuration>ServiceFactory.properties</Configuration>
	</ServiceFactory>
	<Logger>
		<Implementation>PATH\Logger.dll</Implementation>
		<Configuration>Logger.properties</Configuration>
	</Logger>
	<EnvironmentHandler>
		<Implementation>PATH\EnvHandler.dll</Implementation>
		<Configuration>EnvHandler.properties</Configuration>
	</EnvironmentHandler>
</Context>
```

Vidimo lahko, da globalni kontekst kot korenski element v XML-u nima nobenih posebnih atributov, kot je ime ali konfiguracija. Podelementi korenskega elementa so imena skupnih servisov. Vsak servis ima dva razdelka, ki opisujeta njegove lastnosti.

Razdelek `<Implementation>` vsebuje relativno ali absolutno pot do datoteke, ki vsebuje implementacijo komponente. Ponavadi je to dinamična knjižnica .dll na Windows platformi, .so oziroma .sl na Linux/Unix platformah.

`<Configuration>` razdelek pa vsebuje relativno ali absolutno pot do konfiguracijske datoteke za skupni servis.

[7] Pot na datotečnem sistemu, ki je lahko absolutna ali relativna

Pri obeh razdelkih (implementacija in konfiguracija) ime datoteke brez poti pomeni, da se datoteka nahaja v isti mapi kot izvršna datoteka našega programa, ki poganja kompozit.

2.6.2. Večnitnost

V kompozitu je lahko samo ena komponenta, ki jo lahko poženemo. To komponento poimenujemo poganjalna komponenta. Vse ostale komponente v kompozitu so pasivne, upravlja pa jih poganjalna komponenta.

Metodo `Start`, ki je del poganjalne komponente, moramo poganjati asinhrono, ker se mora kompozit odzivati na zunanje signale, kot so `Suspend, Resume` in `Abort.`

Način klicanja metod se navezuje na posamezne implementacije komponent, zato ne moremo imeti nikakršne povezave med komponentami in nitmi, v katerih so pognane. Z drugimi komponentami, ki jih upravlja poganjalna komponenta, pa lahko izvajamo asinhrone klice do metod drugih komponent.

Ker želimo imeti sposobnost suspendiranja in nadaljevanja kompozitov, moramo vse niti, ki so del kompozita (vključno z glavno metodo `Start`), registrirati v kontekstu kompozita.

Zaradi poenostavljenja diplomske naloge smo se odločili, da večnitni kompoziti ne bodo mogoči, vendar bomo metodo `Start`, ki je del poganjalne komponente, poganjali v posebni niti, ki jo bomo registrirali v kontekst kompozita.

2.6.3. Tipizirane lastnosti

Tipizirane lastnosti pomenijo, da se lahko iz XML oblike serializirajo in deserializirajo vrednosti v točno določen tip podatka. Zavoljo enostavnosti bi lahko predpostavljali, da so vse lastnosti tipa `string`, ki bi ga v vsaki implementaciji komponente posebej obravnavali in spreminjali v željen podatkovni tip, vendar to ni najbolj priročna rešitev za končnega uporabnika. Takšna rešitev lahko tudi privede do subtilnih napak. V izogib temu lahko uporabljamo odsevni mehanizem hierarhije objektov, da sam nastavlja vrednosti podatkov v pravilnih tipih brez posebnih »getter« in »setter« metod. Vzemimo primer lastnosti v kompozitu:

```
<sca:property name="m_filename"
type="components::types::CString"> SomeFilename.xml
</sca:property>
```

V takšnem primeru lahko uporabimo pretvorni mehanizem, ki pretvarja vrednosti preko mehanizma prirejanja za nadaljnjo notranjo uporabo:

```
CAdaptor adaptor =
CSerializerEngine::Deserialize(propertyNode);

component.GetTypeInfo().GetPropertyInfo("m_filename").SetValu
e(component, adaptor);
```

Tako imamo mehanizem, ki skrbi, da se vrednosti iz kompozitne specifikacije pravilno preslikajo v pravi podatkovni tip v implementacijah komponent.

2.6.4. Tipizirane reference

Reference so podobne lastnostim v smislu, da so člani C++ implementacij (komponent). Lahko jih tudi nastavljamo preko sistema hierarhije objektov.

Referenca na povezan servis diktira njen tip. Kompozita ne moremo naložiti, če referenca ni tipsko združljiva (isti tip oziroma tip, ki je višje v hierarhiji, naprimer `IntComponent`) s tipom servisa, s katerim je povezana.

Imamo primer servisa

```
<sca:service name="IntComponentPolicy">
      <sca:interface.cpp class="components::policy::IntComponentPolicy"/>
</sca:service>,
```

ki se poveže z referenco v neki drugi komponenti tako:

```
<sca:reference name="m_componentPolicy"/>
```

Tako kot lastnosti moramo tudi reference registrirati v manifestu, tako da lahko sistem upravljanja s tipi dostopa do njih:

```
DEFINE_TYPE_START(SomeComponent)
    …
    DEFINE_REFERENCE(components::policy::CComponentPolicy,
m_componentPolicy)
    …
DEFINE_TYPE_END
```

2.6.5. Množičnost

V SCA kompozitu lahko definiramo množičnost v XML elementu `<sca::reference>`:

```
<sca:reference name="m_componentPolicy" multiplicity=”0..1 or
1..1 or 0..n or 1..n”/>
```

V primeru, da je množičnost 0..n ali 1..n, moramo implementirati vsebovalnik referenc:

```
class CSomeObject : public CBaseObject<CSomeObject,
IntSomeObject>
{
…
private:
      CVector<IntComponentPolicy> m_componentPolicies;[8]
} ;
```

Isto velja za registracijo v manifestu:

```
DEFINE_TYPE_START(SomeObject)
      …
      DEFINE_REFERENCE(CVector<IntComponentPolicy>,
m_componentPolicies)
      …
DEFINE_TYPE_END
```

V primeru, da je množičnost 0..1 ali 1..1, in da je v manifestu registriran vektor referenc namesto posamezne reference, bomo neuspešno naložili kompozit.

V implementacijah komponent lahko svobodno uporabimo vse ali pa izberemo samo posamezne reference, odvisno od njihove notranje logike delovanja.

Množičnost lahko izkoristimo tudi za potrebe upravljanja z napakami. Če je definirana množičnost 1..1 ali 1..n, mora biti referenca povezana z nekim servisom, sicer vemo, da je prišlo do napake.

[8] CVector je implementacija ovitka okoli standardnega C++ tipa std::vector v hierarhiji objektov.

2.6.6. Servisni register (opis delovanja)

Glavno nalogo servisnega registra smo opredelili kot omogočanje komponentam, da najdejo vmesnike servisov, ki so na voljo. Register kot rezultat poizvedbe po servisnem vmesniku vrača seznam knjižnic in njihovih komponent, ki implementirajo servisni vmesnik.

Podatkovno bazo servisnega registra sestavimo iz množice XML datotek. Ena datoteka vsebuje podatke o eni dinamični knjižnici. Te datoteke lahko priložimo pripadajočim dinamičnim knjižnicam v isti mapi ali pa so na ločenem mestu. Podatki v datotekah so sledeči:

- Ime knjižnice, pot in verzija,
- implementacijski razredi komponente ter
- servisni vmesniki, ki jih implementira vsaka implementacija komponente.

Primer XML datoteke servisnega registra:

```
<?xml version="1.0" encoding="utf-8"?>
<library name="lib1" "path=PATH/libname" version="1.0.0">

    <implementation.cpp class="space::name::class1">
        <interface.cpp class="name::space::iservice1"/>
        <interface.cpp class="name::space::iservice1V2"/>
        <interface.cpp class="name::space::iservice2"/>
    </implementation.cpp>

    <implementation.cpp class="space::name::class1"/>
        <interface.cpp class="name::space::iservice1"/>
        <interface.cpp class="name::space::iservice2"/>
    </implementation.cpp>

    <implementation.cpp class="space::name::class2"/>
        <interface.cpp class="name::space::iservice1"/>
        <interface.cpp class="name::space::iservice3"/>
    </implementation.cpp>

</library>
```

Če poženemo poizvedbo po vmesniku, ta vrne enega ali več objektov, knjižnico in informacijo o komponentni implementaciji, ki implementira vmesnik, po katerem poizvedujemo. Komponenta se lahko odloči, katero od implementacij bo uporabila, in tako instancirala ustrezni razred komponentne implementacije skozi instančni mehanizem.

2.7. Kompozit

Kot jedro arhitekture imamo kompozit, ki ga zapišemo v obliki XML v datoteko. V tej datoteki opišemo servise, ki bodo na voljo komponentam, ki bodo zagnane v okviru kompozita.

Primer sestavljenega kompozita:

```
<?xml version="1.0" encoding="utf-8"?>

<!--- Create a composite named Startable -->
<sca:composite xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
name="TestComposite">
  <sca:service name="TestingService" promote="componentTest/Startable"/>

  <sca:component name="componentTest">
    <sca:service name="Startable">
      <sca:interface.cpp class="components::IntStartableComponent"/>
    </sca:service>
    <sca:implementation.cpp class="components::test::CCompPolicyTest"/>
    <sca:property name="m_object" type=" type::IntString[9]">X:\data
\input.txt</sca:property>
    <sca:property name="m_objectResults" type="
type::IntString">100101100</sca:property>
    <sca:reference name="m_componentPolicy" multiplicity="1..1"/>
  </sca:component>

  <sca:component name="ComponentPolicy">
    <sca:service name="IntComponentPolicy">
      <sca:interface.cpp class="component::policy::IntComponentPolicy"/>
    </sca:service>
    <sca:implementation.cpp class="component::policy::CComponentPolicyImpl"
/>
    <sca:property name="m_filename" type=" type::IntString">X:\data
\TestData.xml</sca:property>
  </sca:component>

  <sca:wire source="componentTest/m_componentPolicy"
target="ComponentPolicy/IntComponentPolicy"/>
</sca:composite>
```

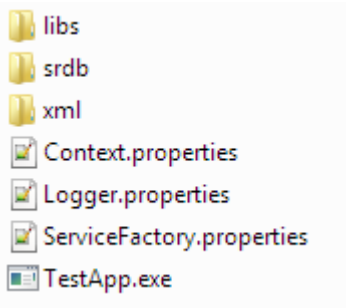
Atribut **type** je obvezen v vseh elementih za lastnosti (properties).
Atribut multiplicity je neobvezen (če ni določen, se privzeto uporabi 0..n).

[9] IntString je vmesnik, definiran v hierarhiji objektov. Lahko ustvarimo svojo implementacijo ali pa uporabimo primitivne standardne tipe.

2.8. Demonstracija uporabe

Za potrebe demonstracije smo ustvarili nekaj testnih komponent in testno aplikacijo, ki požene kompozit.

Celotna struktura na najvišjem nivoju zgleda tako (Slika 5):



Slika 5 Struktura map na najvišjem nivoju

Mapa `libs` vsebuje deljene knjižnice komponent.
Mapa `srdb` (service registry database) vsebuje konfiguracijske datoteke za komponente.
Mapa `xml` vsebuje vse xml datoteke kompozitov.

Datoteka `Context.properties` vsebuje konfiguracijo konteksta.
Datoteka `Logger.properties` vsebuje nastavitve dnevniške komponente.
Datoteka `ServiceFactory.properties` vsebuje nastavitve ServiceFactory komponente.

Datoteka TestApp.exe je izvršilna datoteka programa, ki povezuje celoten sistem.

2.8.1. Testne komponente

Celoten sistem ni sestavljen le iz testnih komponent, ampak smo vanj vključili tudi sistemske komponente (prej imenovane skupni servisi). Komponente smo umestili v `libs` podmapi aplikacije.

Sistemske komponente so sledeče:

- ServiceFactory
- PocoLogger
- ScaContainer
- SerializerBasic
- SerializerFwsTypes

ServiceFactory je komponenta, ki vsebuje poslovno logiko servisnega registra.

PocoLogger je dnevniška komponenta in vsebuje ovojno implementacijo okoli obstoječe rešitve tretje osebe.

ScaContainer je komponenta, v kateri je implementiran Sca vsebovalnik.

SerializerBasic je komponenta, ki vsebuje logiko serializacije enostavnih tipov.

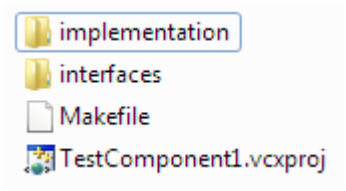
SerializerFwsTypes pa vsebuje logiko serializacije tipov, ki so implementirani v hierarhiji objektov.

Testne komponente so pa:

- TestComponent1
- TestComponent2
- TestComponent3
- TestComponent4

Vsaka izmed njih je precej enostavna komponenta brez zapletene poslovne logike, imajo le osnovno ogrodje, ki jih definira kot komponente.

Opisali bomo primer komponente TestComponent1. Struktura map na korenskem nivoju je razvidna iz slike (Slika 6):



Slika 6 Struktura map TestComponent1

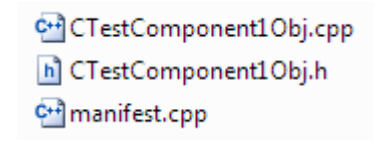
`Implementation` vsebuje implementacijske C++ datoteke.

`Interfaces` vsebuje vmesnike, ki jih TestComponent1 implementira.

`Makefile` je datoteka, ki vsebuje pravila prevajanja na *nix platformah.

`TestComponent1.vcxproj` je Visual Studio 10 projektna datoteka.

V implementacijskem delu imamo sledeče datoteke:

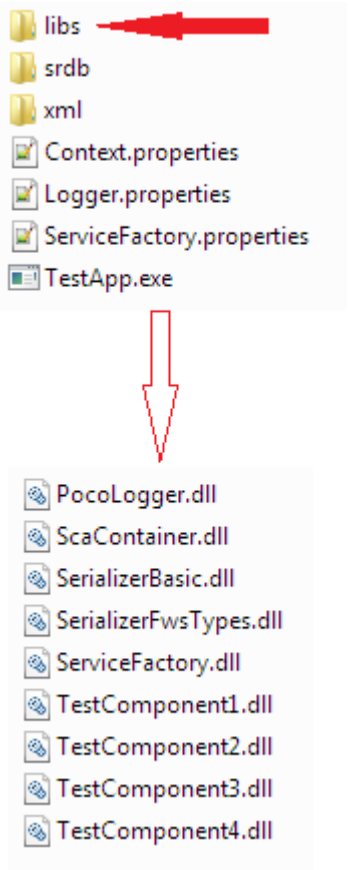


Slika 7 Datoteke testne komponente

V `CTestComponent1Obj.cpp` smo implementirali metode iz `CTestComponent1.h`, kjer so definirane metode in člani komponente.

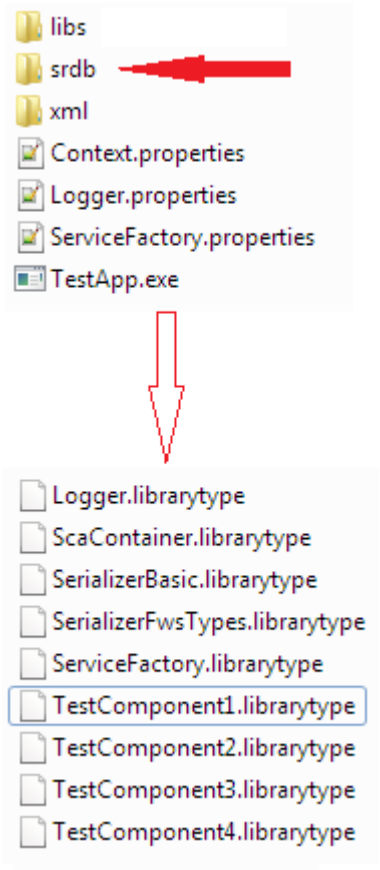
V `Manifest.cpp` smo vpisali definicije tipov in vmesnikov, ki jih komponenta uporabi.

Prevedeno komponento, ki postane dinamična knjižnica, imenovana `TestComponent1.dll (libTestComponent1.so` na *nix sistemih), v ogrodju aplikacije umestimo poleg vseh ostalih komponent v mapo `libs` (Slika 8).



Slika 8 Umeščenost komponent na datotečnem sistemu

Za potrebe `TestComponent1` komponente v `srdb` mapo namestimo konfiguracijsko datoteko `TestComponent1.librarytype` (Slika 9).



Slika 9 Konfiguracijske datoteke komponent

V to datoteko zapišemo, kje se komponenta nahaja na datotečnem sistemu, kako je ime dinamični knjižnici komponente, implementacijski C++ razred, ter katere vmesnike komponenta implementira/podpira:

```
<?xml version="1.0" encoding="utf-8"?>
<library name="TestComponent1" path="libs" version="1.0.0">

    <implementation.cpp class="comps::test::CTestComponent1Obj">
        <interface.cpp class="comps::system::IntComponent"/>
        <interface.cpp class="comps::system::IntStartable"/>
        <interface.cpp class="comps::test::IntBaseTestInterface"/>
        <interface.cpp class="comps::test::IntTestInterface1"/>
        <interface.cpp class="comps::test::IntTestInterface2"/>
    </implementation.cpp>

</library>
```

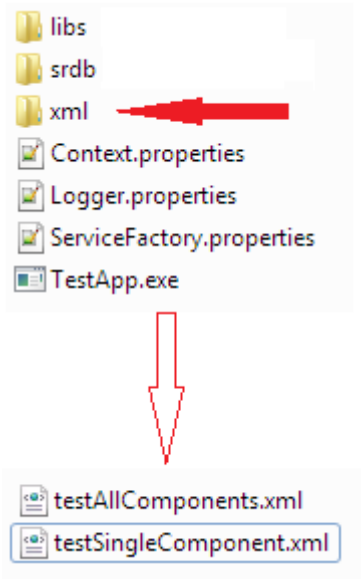
Iz podatkov v tej datoteki lahko vidimo, da se komponenta nahaja na relativni poti »`./libs`« (če črka pogona ali začetna poševnica »/« ni navedena, predpostavimo, da je relativna pot na podlagi trenutnega mesta izvrševalne datoteke na datotečnem sistemu). Dinamična knjižnica komponente se imenuje `TestComponent1.dll`. Z Sca vsebovalnikom prepoznamo platformo ter ustrezno dodamo končnico (.dll/.so/.sl) in morebitno predpono lib na *nix platformah.

Implementacija komponente se nahaja v `CtestComponent1Obj` razredu, kjer smo implementirali naslednje vmesnike:

- IntComponent – sistemski vmesnik, bazični vmesnik (obvezen za vse komponente)
- IntStartable – sistemski vmesnik, bazični vmesnik za poganjalno komponento
- IntBaseTestInterface – bazični vmesnik, napisan za potrebe testiranja (obvezen za vse vse testne komponente)
- IntTestInterface1 – testni vmesnik 1
- IntTestInterface2 – testni vmesnik 2

2.8.2. Primer kompozita

Kompozit smo namestili v `xml` mapo. Tu se lahko nahaja poljubno mnogo kompozitov, vendar za potrebe demonstracije smo napisali samo dva (Slika 10).



Slika 10 Lokacija kompozitnih XML datotek

Za zagon poganjalne komponente `TestComponent1` smo izpostavili servis, imenovan `TestComponent1/Startable`.

```
<?xml version="1.0" encoding="utf-8"?>

<sca:composite xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
name="testSingleComponent">

  <sca:service name="ComponentTestingService"
promote="TestComponent1/Startable"/>

  <sca:component name="TestComponent1">
    <sca:service name="Startable">
      <sca:interface.cpp class="components::system::IntStartable"/>
    </sca:service>
    <sca:service name="BaseTestInterface">
      <sca:interface.cpp class="components::test::IntBaseTestInterface"/>
    </sca:service>
    <sca:service name="TestInterface1">
      <sca:interface.cpp class="components::test::IntTestInterface1"/>
    </sca:service>
    <sca:service name="TestInterface2">
      <sca:interface.cpp class="components::test::IntTestInterface2"/>
    </sca:service>
    <sca:implementation.cpp class="components::test::CTestComponent1Obj"/>
    <sca:reference name="m_componentPolicy" />
    <sca:reference name="m_interface2" />
    <sca:reference name="m_interfaces4" multiplicity="0..n"/>
    <sca:property name="m_length">5</sca:property>
  </sca:component>

</sca:composite>
```

V kompozitu smo opisali vse servise, ki jih komponenta lahko izvaja. V vsakem servisu smo našteli vmesnike, ki so na voljo. Neodvisno od imen servisov in vmesnikov pa smo našteli reference in lastnosti komponente. Kompozit lahko vsebuje več komponent, vendar smo v tem primeru uporabe vpisali samo eno komponento.

Izpostavljenost `TestComponent1/Startable` servisa pomeni, da bomo lahko dostopali do vseh metod, ki implementirajo vmesnike, naštete oziroma registrirane v `Startable` xml razdelku.

V `<sca::interface.cpp>` elementu smo našteli vmesnike, ki so implementirani v servisu. V tem primeru, kjer je izpostavljen `TestComponent1/Startable` servis, bomo imeli dostop do metod, ki so definirane v `components::system::IntStartable` vmesniku.

V `<sca::implementation.cpp>` zapišemo, kateri C++ razred naj bo instanciran.

V `<sca::reference>` elementih naštejemo vse reference na servise drugih komponent.

`<sca::property>` je lastnost, ki smo jo definirali v C++ razredu komponente, katerega je tudi član.

2.8.3. Primer polnega kompozita

Za primer polne uporabe te implementacije arhitekture prilagamo bolj zapleten kompozit, ki vsebuje vse do sedaj naštete implementirane prvine SCA specifikacije:

```
<?xml version="1.0" encoding="utf-8"?>

<!--- Create a composite named testAllComponents -->
<sca:composite xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
name="testAllComponents">
  <sca:service name="ComponentTestingService1"
promote="TestComponent1/Startable"/>
  <sca:service name="ComponentTestingService2"
promote="TestComponent2/BaseTestInterface"/>
  <sca:service name="ComponentTestingService3"
promote="TestComponent3/TestInterface2"/>
  <sca:service name="ComponentTestingService4"
promote="TestComponent4/TestInterface5"/>

  <sca:component name="TestComponent1">
    <sca:service name="Startable">
      <sca:interface.cpp class="components::system::IntStartable"/>
    </sca:service>
    <sca:service name="BaseTestInterface">
      <sca:interface.cpp class="components::test::IntBaseTestInterface"/>
    </sca:service>
    <sca:service name="TestInterface1">
      <sca:interface.cpp class="components::test::IntTestInterface1"/>
    </sca:service>
    <sca:service name="TestInterface2">
      <sca:interface.cpp class="components::test::IntTestInterface2"/>
    </sca:service>
    <sca:implementation.cpp class="components::test::CTestComponent1Obj"/>
    <sca:property name="m_length">5</sca:property>
    <sca:property name="m_cinteger">-55</sca:property>
    <sca:property name="m_cuinteger">53432</sca:property>
    <sca:property name="m_cstring">Hello</sca:property>
    <sca:property name="m_cipaddress">192.168.0.1</sca:property>
    <sca:property name="m_ctime">2011-2-2 13:59:38.121134 GMT</sca:property>
    <sca:property name="m_ctimestamp">2012-1-18 10:08:33.280146
GMT</sca:property>
    <sca:property name="m_ctimespan">2d 12:30:10.123456</sca:property>
    <sca:reference name="m_componentPolicy"/>
    <sca:reference name="m_interface2"/>
    <sca:reference name="m_interfaces4" multiplicity="0..n"/>
  </sca:component>

  <sca:component name="TestComponent2">
    <sca:service name="BaseTestInterface">
      <sca:interface.cpp class="components::test::IntBaseTestInterface"/>
    </sca:service>
    <sca:service name="TestInterface3">
      <sca:interface.cpp class="components::test::IntTestInterface3"/>
    </sca:service>
    <sca:service name="TestInterface4">
      <sca:interface.cpp class="components::test::IntTestInterface4"/>
    </sca:service>
    <sca:implementation.cpp class="components::test::CTestComponent2Obj"/>
    <sca:property name="m_bool">false</sca:property>
    <sca:property name="m_int">5</sca:property>
    <sca:property name="m_short">-55</sca:property>
    <sca:property name="m_long">53432</sca:property>
```

```
    <sca:property name="m_char">12</sca:property>
    <sca:property name="m_wchar">30</sca:property>
    <sca:property name="m_uint">5</sca:property>
    <sca:property name="m_ushort">322</sca:property>
    <sca:property name="m_ulong">4234</sca:property>
    <sca:property name="m_uchar">12</sca:property>
    <sca:reference name="m_interface1"/>
    <sca:reference name="m_interfaces2" multiplicity="0..n"/>
  </sca:component>

  <sca:component name="TestComponent3">
    <sca:service name="Startable">
      <sca:interface.cpp class="components::system::IntStartable"/>
    </sca:service>
    <sca:service name="BaseTestInterface">
      <sca:interface.cpp class="components::test::IntBaseTestInterface"/>
    </sca:service>
    <sca:service name="TestInterface2">
      <sca:interface.cpp class="components::test::IntTestInterface2"/>
    </sca:service>
    <sca:service name="TestInterface4">
      <sca:interface.cpp class="components::test::IntTestInterface4"/>
    </sca:service>
    <sca:implementation.cpp class="components::test::CTestComponent3Obj"/>
    <sca:reference name="m_interface3"/>
    <sca:reference name="m_interface4"/>
    <sca:reference name="m_interface5"/>
  </sca:component>

  <sca:component name="TestComponent4">
    <sca:service name="BaseTestInterface">
      <sca:interface.cpp class="components::test::IntBaseTestInterface"/>
    </sca:service>
    <sca:service name="TestInterface5">
      <sca:interface.cpp class="components::test::IntTestInterface5"/>
    </sca:service>
    <sca:implementation.cpp class="components::test::CTestComponent4Obj"/>
    <sca:reference name="m_interface3"/>
    <sca:reference name="m_interface4"/>
  </sca:component>

  <sca:wire source="TestComponent1/m_interface2"
target="TestComponent3/TestInterface2"/>
  <sca:wire source="TestComponent2/m_interface1"
target="TestComponent1/TestInterface1"/>
  <sca:wire source="TestComponent3/m_interface3"
target="TestComponent2/TestInterface3"/>
  <sca:wire source="TestComponent3/m_interface4"
target="TestComponent3/TestInterface4"/>
  <sca:wire source="TestComponent3/m_interface5"
target="TestComponent4/TestInterface5"/>
  <sca:wire source="TestComponent4/m_interface3"
target="TestComponent2/TestInterface3"/>
  <sca:wire source="TestComponent4/m_interface4"
target="TestComponent3/TestInterface4"/>

</sca:composite>
```

3. Sklepne ugotovitve

SCA specifikacija je obsežnejša, kot je je bilo implementirano v tem diplomskem delu. Tukaj smo obravnavali samo glavnino uporabnosti SCA arhitekture, ki bi lahko nadomestila klasičen model pisanja in posodabljanja programov.

Nekaj podrobnosti je bilo tudi izpuščenih iz naloge, na primer implementacija luščenja XML datotek oziroma uporaba tretjega vmesnika API za luščenje XML oblike, saj se nam to ni zdel primeren del naloge, ki bi ga izpostavljali, saj bi tako le zameglili večjo sliko, ki jo predstavlja implementacija SCA specifikacije. Prav tako se iz istih razlogov nismo spuščali v podrobno delovanje nekaterih mehanizmov, kot so tovarna servisov, servisni register in serializacija tipov iz XML oblike.

Izpuščeno je bilo tudi opisovanje nalagalnega mehanizma za deljene knjižnice, saj je to odvisno od želje implementatorja, na kakšen način želi nalagati knjižnice. Najenostavneje je uporabiti standardni C++ vmesnik API, ki je zadovoljivo zadosten za pokritje večine platform, na katerih bi radi poganjali določen sistem.

Sam sistem bi lahko bil razširjen tako, da bi podpiral več poganjalnih komponent, več vzporednih globalnih kontekstov in podobno.

Končna ugotovitev je, da je tovrstna minimalna implementacija zadostna za ustvarjanje kompleksnejših aplikacijskih sistemov. Za namene posodabljanja pa lahko programer posodablja posamezne deljene knjižnice v sistemu, ne da bi kakorkoli manipuliral z izvršno kodo glavnega programa. Pogoj je le, da implementirana komponenta sledi definiranim vmesnikom.

4. Literatura

- (2013) Dokumentacija o polimorfizmu v C++. Dostopno na: http://www.cplusplus.com/doc/tutorial/polymorphism/
- (2013) Domača stran OASIS Service Component Architecture. Dostopno na: http://www.oasis-opencsa.org/sca
- (2013) SCA assembly. Dostopno na: http://docs.oasis-open.org/opencsa/sca-assembly
- (2013) XML specifikacija. Dostopno na: http://www.w3.org/TR/REC-xml/