NTNU
Kunnskap for en bedre verden

Institutt for datateknikk og informasjonsvitenskap

Eksamensoppgave i TDT4250
Avansert programvaredesign

Faglig kontakt under eksamen: Hallvard Trætteberg

Tlf.: 91897263

Eksamensdato: 16. desember

Eksamenstid (fra-til): 9.00-13.00

Hjelpemiddelkode/Tillatte hjelpemidler: C

Bestemt, enkel kalkulator tillatt.

Annen informasjon:

Oppgaven er utarbeidet av faglærer Hallvard Trætteberg og kvalitetssikret av John Krogstie.

Målform/språk: Bokmål

Antall sider: 3

Antall sider vedlegg: 2

Kontrollert av:

_____

Dato Sign

Dersom du mener at opplysninger mangler i en oppgaveformulering, gjør kort rede for de antagelser og forutsetninger som du finner nødvendig.

Del 1 – Modell-basert utvikling (35%)

a) Hva er forskjellen mellom modeller av typen CIM og PIM? Vil modeller laget med ecore være av typen CIM eller PIM, eller er det ikke gitt på forhånd?

b) Hva er forholdet mellom en instans, modell og meta-modell? Illustrer modell-nivåene M0 til M3 med eksempler.

c) Transformasjoner klassifiseres på flere måter. Forklar følgende kategoriseringer:
   - M2M – M2T
   - Exogenous – endogenous
   - In-place – out-place

d) EMF-rammeverket bruker flere typer transformasjoner. Klassifiser følgende transformasjoner iht. kategoriene fra c):
   - Fra ecore til genmodel, altså steget før en kan generere Java-kode fra en ecore-modell.
   - Fra genmodel til Java-kode, altså når en (re)genererer Java-kode fra genmodel.

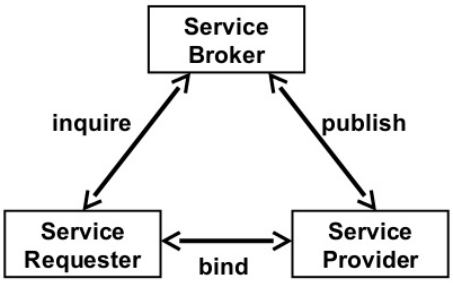
e) Hvordan er den generelle virkemåten til ATL-baserte M2M-transformasjoner? Beskriv forskjeller og likheter mellom slike transformasjoner og *graf*-transformasjoner?

Del 2 – Komponent-basert konstruksjon (35%)

a) OSGi-moduler (bundles) definerer to typer avhengigheter (begge typer deklareres i MANIFEST.MF), hvilke? Hva er fordeler og ulemper med disse to typene?

b) Beskriv livssyklusen til OSGi-moduler (bundles), med fokus på sammenhengen mellom tilstander og avhengigheter mellom moduler.

c) Forklar følgende figur, med fokus på hvordan et system av kjøretidskomponenter rigges opp:



d) Hva er Dependency Injection (DI) og hvordan er det brukt av OSGi Declarative Services, f.eks. i øvingsprosjektet? Hvilke fordeler og ulemper ser du ved bruk av DI på denne måten?

e) REST API-er er ofte tett koblet til en underliggende datamodellen. På hvilken måte? Hva er problemet med denne tette koblingen, og hvordan ble dette håndtert i øvingsprosjektet?

Del 3 – DSL-design (30%)

Med OSGi Declarative Services (DS) så deklareres kjøretidskomponenter i egne XML-filer (som det henvises til fra MANIFEST.MF). Du finner eksempler på slike fra øvingsprosjektet i vedlegget **Sample component definitions**. Slike filer er tungvinte å skrive og det oppstår lett feil som er vanskelige å finne, f.eks. ved å feilstave klasse- og metodenavn i **reference**-elementet. En får dessuten ikke *validert* egendefinerte komponentegenskaper deklarert med **property**-elementet. F.eks. fikk mange grupper feil i deklarasjonen av deres app-komponent (tilsvarende **GameApp**-komponenten vist i vedlegget), fordi **EngineAppComponent.eClass**-egenskapen hadde et spesielt format som ikke ble sjekket av Eclipse.

I denne oppgaven skal du lage et DSL som er ment å 1) gjøre det raskere å deklarere komponentene og 2) gjøre det lettere å unngå og rette slike feil som er nevnt over.

a) Lag en objekt-orientert modell som fanger opp informasjonen som en ønsker å representere i disse XML-filene. Tenk spesielt på hva modellen må inneholde for å kunne nå målene 1) og 2) over for øvingsprosjektet. Husk samtidig at instanser av modellen må kunne oversettes til det eksisterende XML-formatet (d, under). Inkluder klasser, datatyper, attributter og referanser (inkl. aggregeringer) med multiplisitet, altså den samme informasjon en trenger for å lage en komplett ecore-modell som grunnlag for en DSL. Velg selv om du vil tegne et diagram og/eller bruke tekst. I vedleggene **Ecore data types** og **Ecore classes** finner du ecore-typer du kan bruke i modellen.

b) Beskriv en syntaks for et DSL for komponent-deklarasjoner basert på modellen fra a), og vis eksempler basert på vedlegget **Sample component definitions**. Forklar hvordan du vil definere syntaksen formelt, om du nå velger en tekst-basert syntaks med Xtext eller diagram-basert syntaks med Sirius.

c) Beskriv hvordan funksjoner/verktøy knyttet til (editoren for) DSL-en kan hjelpe deg å nå målene 1) og 2) over.

d) Forklar hvordan du vil lage en transformasjon fra DSL-modellen til XML-formatet.

NTNU – Trondheim
Norwegian University of
Science and Technology

Department of computer and information science

Examination paper for TDT4250 Advanced Software Design

Academic contact during examination: Hallvard Trætteberg

Phone: 91897263

Examination date: 16. December

Examination time (from-to): 9:00-13:00

Permitted examination support material: C

Specific, simple calculator is allowed.

Other information:

This examination paper is written by teacher Hallvard Trætteberg, with quality assurance by John Krogstie.

Language: English

Number of pages: 3

Number of pages enclosed: 2

Checked by:

_____

Date Signature

If you feel necessary information is missing, state the assumptions you find it necessary to make.

Part 1 – Model-based development (35%)

a) What is the difference between CIM and PIM type of models? Will ecore models be of type CIM or PIM, or can't it be decided in advance?

b) What is the relation between an instance, a model and a meta-model? Illustrate the model levels M0 to M3 with examples.

c) Transformations can be classified in several ways. Explain the following categories:
   - M2M – M2T
   - Exogenous – endogenous
   - In-place – out-place

d) The EMF framework uses several kinds of transformations. Classify the following transformations according to the categories from c):
   - From ecore to genmodel, i.e. the step before you can generate Java code from an ecore model.
   - Fra genmodel to Java code, i.e. when you (re)generate Java code from genmodel.

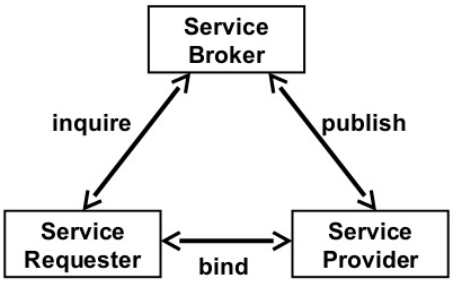
e) How do ATL-based M2M transformations work in general? Describe similarities and differences between such transformations and *graph* transformations.

Part 2 – Component-based design (35%)

a) OSGi modules (bundles) define two types of dependencies (both are declared in MANIFEST.MF), which ones? What are advantages and disadvantages with these to types?

b) Describe the life-cycle of OSGi modules (bundles), with focus on the relation between states and module dependencies.

c) Explain the following figure, with focus on how a system of components is rigged:



d) What is Dependency Injection (DI) and how is it used by OSGi Declarative Services, e.g. in the exercise project? What advantages and disadvantages do you see for such use of DI?

e) REST APIs are often tightly coupled to an underlying data model. In what way? What is the problem with this tight coupling, and how is this handled in the exercise project?

Del 3 – DSL-design (30%)

With OSGi Declarative Services (DS) components are declared in separate XML files (that are referenced from MANIFEST.MF). You find examples of such from the exercise project in the **Sample component definitions** appendix. Such files are cumbersome to write and errors are easily introduced and difficult to find, e.g. by misspelling class and method names in the **reference** element. In addition, there is no *validation* of custom component properties declared with the **property** element. E.g., a lot of groups made mistakes in the declaration of their app component (corresponding to the **GameApp** component shown in the appendix), because the **EngineAppComponent.eClass** property had a special format that wasn't checked by Eclipse.

In this part you will make a DSL that is meant to 1) make it quicker to declare components and 2) easier to avoid and fix mistakes like those mentioned above.

a) Make an object-oriented model that captures the information you wish to represent in these XML files. In particular, consider what the model must include to achieve the goals 1) and 2) over for the exercise project. And keep in mind that it must be possible to convert instances of the model to the existing XML format (d). Include classes, data types, attributes and references (incl. aggregations) with multiplicity, i.e. the same information you need to make a complete ecore model as the basis for a DSL. You're free to decide to use a diagram and/or text. In the appendix **Ecore data types** and **Ecore classes** you'll find ecore types you can use in your model.

b) Describe a syntax for a DSL for component declarations based on the model from a), and show examples based on the appendix **Sample component definitions**. Explain how you'll define the syntax formally, whether you choose a textual syntax with Xtext or diagram-based syntax with Sirius.

c) Describe how functions/tools related to (the editor for) the DSL can help you achieve goals 1) and 2) above.

d) Explain how you'll make a transformation from the DSL model to the XML format.

Vedlegg/Appendix

Sample component definitions.

```
<scr:component name="no.hal.pg.runtime.engine.util.PlayerReferenceHandler">
   <implementation class="no.hal.pg.runtime.engine.util.PlayerReferenceHandler"/>
   <service>
      <provide interface="no.hal.pg.runtime.engine.IReferenceProvider"/>
      <provide interface="no.hal.pg.runtime.engine.IReferenceResolver"/>
   </service>
</scr:component>

<scr:component name="no.hal.pg.runtime.engine.http.EngineAppEndPointProvider">
   <implementation class="no.hal.pg.runtime.engine.http.EngineAppEndPointProvider"/>
   <reference bind="setHttpService" name="HttpService" cardinality="1..1"
       interface="org.osgi.service.http.HttpService" unbind="unsetHttpService"/>
   <reference bind="addEngine" cardinality="0..n"
       interface="no.hal.pg.runtime.engine.IEngine" name="Engine" unbind="removeEngine"/>
   <reference bind="addEngineApp" name="EngineApp" cardinality="0..n"
       interface="no.hal.pg.runtime.engine.http.IEngineApp" unbind="removeEngineApp"/>
</scr:component>

<scr:component name="no.hal.pg.runtime.engine.web.GameApp">
   <implementation class="no.hal.pg.runtime.engine.http.EngineAppComponent"/>
   <service>
      <provide interface="no.hal.pg.runtime.engine.http.IEngineApp"/>
   </service>
   <property name="IEngineApp.name" type="String" value="GameApp"/>
   <property name="EngineAppComponent.eClass" type="String"
       value="platform:/plugin/no.hal.pg.runtime/model/pg-runtime.ecore#Game"/>
   <property name="IEngineApp.displayName" type="String" value="Game app"/>
   <property name="EngineAppComponent.main" type="String"
       value="/web/GameApp.html"/property>
   <property name="EngineAppComponent.resourceNames" type="String"
       value="/web"></property>
   <property name="EngineAppComponent.aliasPathFormat" type="String" value="/web"/>
</scr:component>
```

Ecore data types

Name	Java type	Description
EInt	int	int values
EIntegerObject	Integer	Integer objects
EDouble	double	double values
EDoubleObject	Double	Double objects
EBoolean	boolean	boolean values
EBooleanObject	Boolean	Boolean objects
Estring	String	String objects
EJavaClass	Class<?>	Java Class objects

Ecore classes

Name	Description
EPackage	An ecore package. Has a name and id (URI) and contains EClassifiers.
EClassifier	Superclass of EClass and EDataType
EClass	Ecore's class concept, contains EStructuralFeatures
EDataType	Wraps ordinary Java types, so they can be used in Ecore models
EStructuralFeature	Superclass of EAttribute and EReference. Has a name, a type and multiplicity.
EAttribute	A structural feature with an EDataType as the type.
EReference	A structural feature with an EClass as the type.