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?

A CIM (Computation Independent Model) is used during analysis of the problem domain (problem description and requirements) and does not consider issues specific for the design and implementation of a system. A PIM is a design model where software issues are considered, but it is still independent of the actual platform for realizing the software.

Ecore can be used for both CIM and PIM, but is more relevant for PIM, since it includes concepts that design-oriented (e.g. data types) and excludes many constructs relevant for CIM, e.g. association classes.

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

Classes in an object-oriented model can be instantiated, e.g. from a Person class you can create instances corresponding to specific persons like yourself. We say that a class is a meta-level above the instance, M1 and M0, respectively. The model that defines the constructs used for defining classes (and the other elements in the) in M1, is called a meta-model, since the it is at a meta-level above, M2, in this case. Typical elements in a meta-model for object-oriented models are Class, Association and Attribute. The meta-model is itself a model, so the constructs used for defining the Class, Association and Attribute concepts are themselves modeled in the meta-meta-model, i.e. M3. Sometimes M3 and M2 is actually the same language, e.g. Ecore is its own meta-model, and sometimes they are different but similar, e.g. UML is modeled using MOF, which is a subset of UML class models used for general meta-modeling.

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

M2M – model to model transformations, where the target of the transformation is another model, vs. M2T – model to text transformation, where the target of the transformation is text (a string).

Exogenous, where the target language is different from the source, vs. endogenous, where the target language is the same as the source.

In-place, where the target model is the same model, i.e. it is modified in-place, vs. out-place, where the target model is created as a result of executing the transformation.

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.

Ecore to genmodel is **M2M**, since both are models, **exogenous**, since the are instances of different modeling languages, and out-place, since the genmodel is created.

Genmodel to Java code is **M2T**, since Java code is text, **exogenous**, since they are (instances of) different (modeling languages) and out-place, since the Java code is created. If you consider that the Java code is merged with the existing code, meaning that the transformation is from genmodel and Java code to Java code, the transformation becomes partly endogenous and in-place.

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

ATL is based on rules for mapping source objects to target objects, i.e. if you have a certain source object of some type and satisfying certain conditions, you want to create a certain target object of some type and with certain properties. Target objects are assumed linked to other target objects, and the execution of the whole transformation takes care to construct the resulting graph in a proper order, so that the target model is complete and connected.
A graph transformation is more general, in that it is a mapping from a source graph fragment pattern to a target graph fragment. The source pattern is matched to source graph fragments, and the target graph fragment is constructed from the binding between pattern and actual objects.

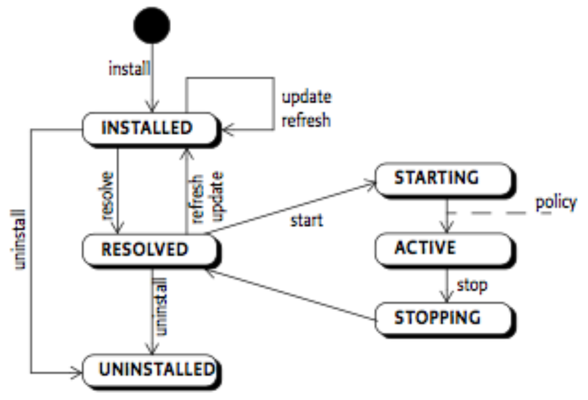
Part 2 – Component-based design (35%)

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?

(Depending means requirements for resolving a bundle.) A bundle can depend on other bundles (with Require-Bundle), meaning that the other bundles must be present and resolved for this bundle to be resolved. A bundle can depend on packages (with Import-Package), meaning that for each of the package there must be a bundle present and resolved that exports the package.
Requiring bundles makes it easier to assemble (provision) the actual system, since you explicitly have the bundle names (and versions), but you have less freedom in repackaging the software.
Requiring packages makes more it easier to repackage the software into different bundles without affecting the resolving process, but makes provisioning more difficult, since you must know more about the content of each bundle. In general, requiring packages is more robust, and with tool support it needn't be more difficult in practice.
Think of package as the interface of a part of the system, with the bundles as the implementation. The advantages and disadvantages are similar to that of using interfaces vs. implementation types.

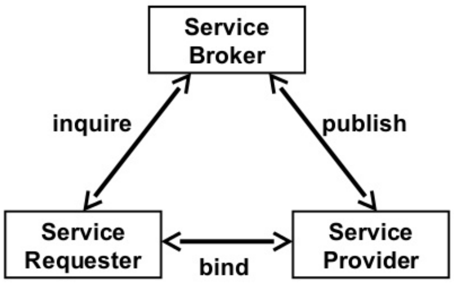
b) Describe the life-cycle of OSGi modules (bundles), with focus on the relation between states and module dependencies.

Figure 4.28 *State diagram Bundle*



The main states are **installed**, **resolved** and **active**. A bundle is first installed, and if its dependency requirements are met and the corresponding bundles are resolved, it is itself resolved. A resolved bundle may be started, and while that process is ongoing (classes are loaded and the activator is running) the bundle is in the intermediate **starting** state. Once the starting process is completed the bundle is **active**. The bundle may for various reasons be stopped, e.g. some dependent bundle is stopped, and during that process it is **stopping**. It will then be resolved or end up being just installed, if the requirements are no longer met. A bundle may of course also be explicitly uninstalled.

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



In a component-based system, components don't know about each other in advance, but are assembled into a graph based on dependencies. The graph is dynamic, i.e. may vary over time. The figure the components managing this process. A component, the Service Provider, tells a broker (publish) that it is ready to receive requests, but sending a publish message. When another component, the Service Requester, needs a particular service, it asks the Service Broker about potential providers (inquire), and when one is found and selected, the Requester and Provider are bound (bind) together, e.g. get direct references to each other. Later (and not indicated in the figure), when a Provider may withdraw and will then be unbound from other components.

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?

DI is the automatic assembly of objects into a graph based on metadata or annotations about dependencies. It relieves the programmer from writing this by hand, and in general DI systems make it easier to support different configurations, e.g. for testing. Since the assembly process is less transparent and deterministic, it may make the code more difficult to write and debug. OSGi Declarative Services (DS) is a DI mechanism that loads the component dependency metadata from XML files that are referenced in MANIFEST.MF. It's also possible to use annotations in the Java code. The metadata includes references to classes and methods that are used for instantiation and binding (graph assembly). Components like the various ITaskProvider and ITaskServiceProvider service implementations were bound to the framework components using OSGi DS.

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?

The REST URLs can be considered paths that navigate through and select runtime objects in a graph based on (names in) the data model. The whole graph is exposed, and there is poor support for encapsulation (different visibility for external request) or role-based views (different visibility based on roles). In the exercise project, encapsulation was supported by wrapping the runtime objects in services, that selectively exposed part of the runtime object graph. The services could remove, expose, modify or add properties to the actual runtime objects.

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.

The model must capture all the information that the XML format contains, and should have better support for custom properties types (that are mapped to string when transforming to XML). This will allow reaching goals 1) by supporting smarter editing and completion and 2) validation and correction.

```
class Component {
	attr EString name;
	EJavaClass implementationClass;
	EJavaClass[*] services;
	ComponentReference[*] componentReferences;
	Property[*] properties;
}

class ComponentReference {
	attr EString name;
	EJavaClass service;
	Cardinality cardinality;
	attr MethodRef bind;
	attr MethodRef unbind;
}

datatype MethodRef wraps EString; // can check for valid syntax and existence

class Cardinality {
	attr EInt lowerBound default -1;
	attr EInt upperBound default -1;
}
```

```
abstract class Property {
        op EString getType(); // computes XML attribute value, implemented in all subclasses
        op EString asString(); // computes XML attribute value, implemented in all subclasses
        attr EString name;
}
class StringProperty {
        attr EString stringValue;
}

… other standard Property subclasses here…

class EClassProperty {
        ref EClass classRef;
}
```

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.

We choose to make a textual DSL. Here's an example showing all elements:

```
component no.hal.pg.runtime.engine.util.PlayerReferenceHandler
   implemented-by no.hal.pg.runtime.engine.util.PlayerReferenceHandler
   providing
      no.hal.pg.runtime.engine.IReferenceProvider,
      no.hal.pg.runtime.engine.IReferenceResolver
   reference HttpService[1..1] to org.osgi.service.http.HttpService
      with setHttpService (unsetHttpService)
   reference Engine[0..] to no.hal.pg.runtime.engine.IEngine
      with addEngine (removeEngineApp)
   string IEngineApp.name = "GameApp"
   eClass EngineAppComponent.eClass = "platform:/plugin/no.hal.pg.runtime/model/pg-
runtime.ecore" # Game
```

Below is an Xtext grammar (not written or tested in Eclipse). Many details are beyond what is expected, in particular the use of datatype rules (QName, MethodRef and EClassRef).

```
Component:
        name = QName ‘implemented-by’ implementationClass  = QName
        (‘providing’ services += QName (‘,’ services += QName)*)?
        (componentReferences += Reference)*
        (properties += Property)*
        ;

QName = ID (‘.’ ID)+; // so-called datatype rule, for qualified names and EJavaClass attributes

Reference:
        ‘reference’ name = QName cardinality = Cardinality
        ‘to’ service = QName ‘with’ bind = MethodRef (‘(‘ unbind = MethodRef ‘)’)?
        ;
```

```
MethodRef: QName; // method references

Property: StringProperty | … | EClassProperty; // disjoint type rules
StringProperty: ‘string’ name = QName ‘=’ value = STRING;
EClassProperty: ‘eClass’ name = QName ‘=’ [EClass | EClassRef]; // reference EClass, typically in separate file using special syntax
EClassRef: STRING ‘#’ ID; // syntax of EClass reference, must be handled by custom Xtext code
```

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

The syntax is less verbose than the corresponding XML, and with Xtext’s built-in completion both 1) and 2) are improved. You can speed up the editing process (1) even more by providing custom completion for EJavaClass attributes utilizing the project class path, and for EClass references (1) by locating Ecore models. Validation will indicate unresolved references to Java classes and and EClasses, and completion will help fix mistakes (2).

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

This is an M2T transformation where you should use a custom template language like Acceleo or a generic language with support for string templates like Xtend. The transformation will typically have a ‘generate’ method for each model class that adds strings to the output based on the corresponding object’s contents. The ‘generate’ method for the containers will call ‘generate’ methods on the contained objects. The Property class’ method getType and asString will be used for Property objects. Utility methods will ensure the output is proper XML, e.g. for quoting/encoding attribute values.

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.