Practical Course “Model-Driven Software Development” Summer Term 2023

Practice Sheet 1

Dr.-Ing. Erik Burger, Dr. rer. nat. Max Kramer

Due: 10 May 2023

Exercise 1: Component-Based Systems

a) Develop a metamodel that can one day be used to design, illustrate and generate code for component-based system architectures. In particular it should later be possible to design graphical editors for it in order to depict instances of it according to four different view types. Furthermore, model instances of your metamodel shall one day be used to generate structural code stubs for different platforms and programming languages from it. For now, you should make sure that your metamodel does not interfere with these future goals and that it is complete. *In particular, every concern that appears in the text either has to be directly modelled in your metamodel or it has to be ensured using OCL constraints.* If you add information or constraints to your metamodel that were not mentioned in the text, then you need to have a good reason for that, which you have to write down in a separate text file. You can create one or multiple `.ecore` files to realize the metamodel.

    Component-based systems can be designed using three different view points consisting of four different view types (see Figure 1): The system-independent view point has a single *repository* view type that shows all components and interfaces that may be reused within multiple systems. All remaining view points depict system-specific information. The assembly view point has a single *assembly* view type, which shows how components are instantiated in a given system and how these instances are connected. Two view types are part of the last view point for deployment: the *environment* view type depicts all containers and the links between them; the *allocation* view type specifies which component instances of the assembly view type are allocated on which containers of the environment view type.

    A repository contains interfaces and components, which may be reused for multiple systems. An interface has a name and consists of signatures, which have a name and a return type and may have parameters that are typed and named. Void is a possible return type in addition to the types that are allowed for parameters. These parameter types are complex types and the following simple types: `Boolean`, `Char`, `Date`, `Double`, `Float`, `List`, `Int`, `Long`, `Map` and `String`. Components have a name and provide interfaces to other components. If

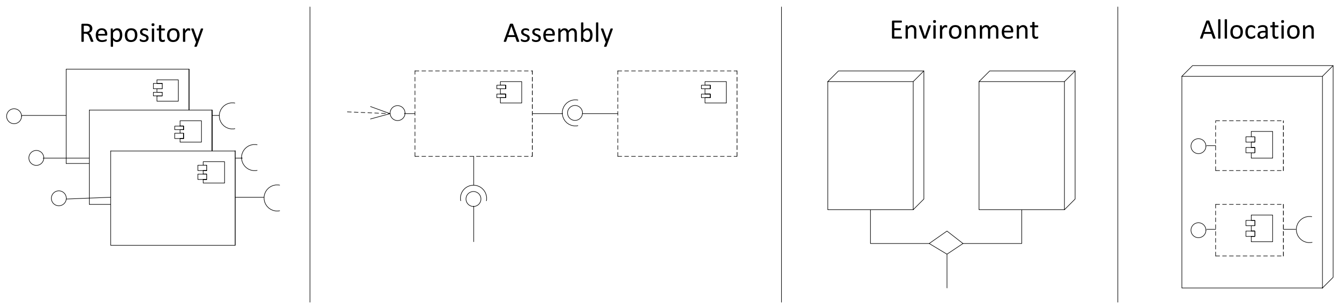

Figure 1: View types of component-based systems

a component provides an interface it provides services corresponding to the signatures of the provided interface. It is possible that a component provides a service that corresponds to multiple, identical signatures of different interfaces. In order to realize services, components can require interfaces.

For the description of how calls to provided services are propagated to required services, a component can contain a behaviour description. The behaviour description can contain internal actions, external calls to required services, loops, anc branches.

Instances of components are system-specific and are called assembly contexts. They are the main artefact of the assembly view type. An assembly context has a name and instantiates a component. Composite components are components that encapsulate assembly contexts of other components. A system also encapsulates assembly contexts and provides at least one interface. Provided and required interfaces of a component result in provided and required roles for the assembly contexts of the component. Delegation connectors can appear in two settings: a) either they link a provided interface of a composite component or system to a provided role of an assembly context. b) or they link a required role of an assembly context to a required interface of a composite component or system. Assembly connectors link a provided role of an assembly context to a required role of another assembly context.

The environment consists of named containers and named links that connect at least two containers. The allocation of each system is specified using allocation contexts. An allocation context specifies which top-level assembly context of a system is allocated on which container of the corresponding environment. Note that assembly contexts that are not a direct part of a system but reside within a composite component cannot be allocated on containers. They are implicitly allocated within the allocation contexts that implement their hosting composite component. This implicit allocation of assembly contexts that reside within composite components does not have to be modelled. If the roles of two assembly contexts are connected by an assembly connector they either have to be allocated on the same container or the containers on which they are allocated have to be linked.

This domain description does not explain what the individual elements of component-based systems are. It is not mentioned, for example, that a component can directly be reused without any knowledge of the internal realization. Further information on elements of component-based systems is, for example, available from the wikipage of the lecture on component-based software architectures[1]. This source of

[1] `https://sdqweb.ipd.kit.edu/wiki/Vorlesung_Komponentenbasierte_Software-Architektur_SS15`
User: stud, Password: kbswa_sose_2015

information on component-based modelling languages and other sources that you may find can help you to understand the basic concepts. *Nevertheless, the information given in this exercise has absolute priority for your metamodel in cases of conflicting information.*

b) You ensured completeness of your metamodel in the previous exercise. Now it is time to increase the quality of your metamodel. In order to refactor your metamodel copy your existing metamodel. Explain in a separate text file for every executed refactoring step your intent and give some short rationale. In your improved version try to achieve the following goals: Concepts that are completely identical for multiple elements should only be specified once. At the same time the parts of your metamodel that represent different view types for component-based systems should be as independent of each other as possible. Related elements should be *grouped into packages* or subpackages. If concepts reoccur in a similar but not identical manner, try to separate the variable parts from the fixed parts in order not to repeat yourself. At the same time *avoid to model different concepts the same way and avoid to model identical concepts differently. Separate metaclasses of different view-types* from each other wherever this is appropriate in order to avoid unnecessary coupling. Decide thoughtfully whether references shall be *navigable* in both directions, which elements shall *contain* others, whether inheritance or references shall be used and which metaclasses shall be *instantiable.* Take care: the order in which the concerns appear in the domain description may not be aligned to the view types. Some concerns require the knowledge of other concerns so that they may be mentioned in text paragraphs that belong to another view type.

c) Generate an editor for your metamodel and use it in order to create models that represent the system depicted in Figure 2. The elements marked in grey are not part of the system. They are only shown in order to illustrate the purpose of the system. Note that the view of the figure does not correspond to a view type mentioned in the domain description. It is a compact mix that simplifies many of the concerns mentioned above. Nevertheless, you should *separate elements of different view types* and create *four models that correspond to the four view types.* These models should contain a system, two containers, a link, five atomic components, a composite component, four interfaces, seven signatures, nine services, six assembly contexts, six provided roles, five required roles, three delegation connectors, four assembly connectors and three allocation contexts. Check in all your model instances. It may be that you have to adapt your metamodel to create the instance. If this is the case, check the modified ecore files into the svn and shortly describe your changes in a separate text file.

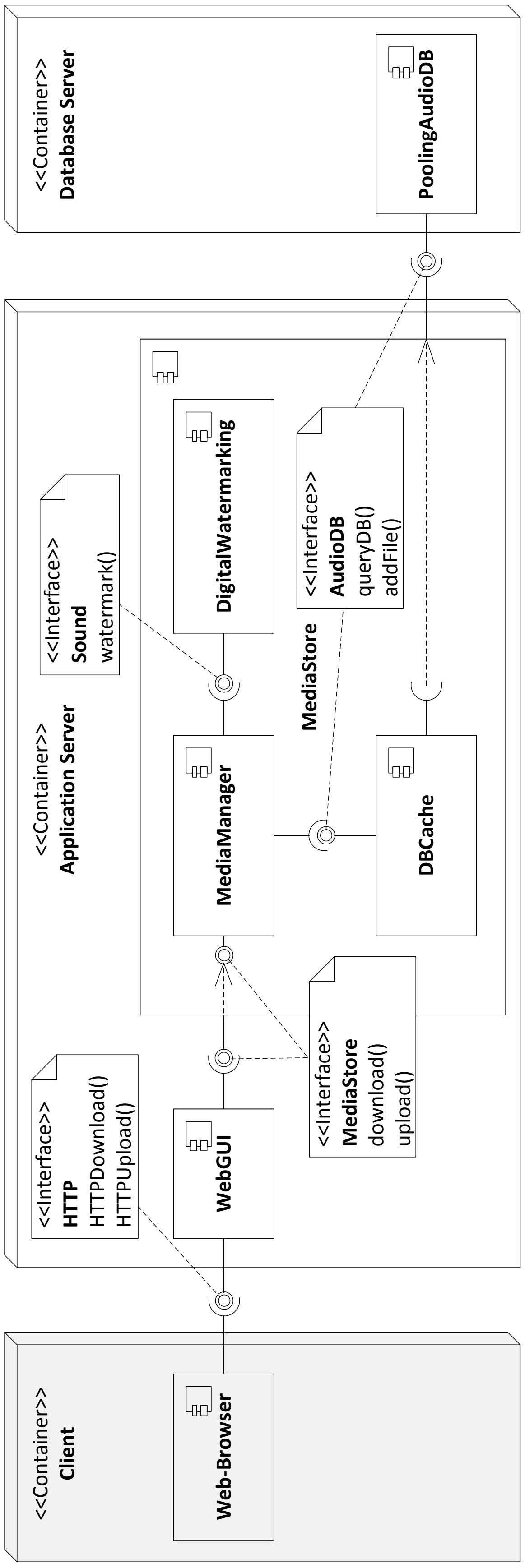

Figure 2: A component-based system for media files based on Becker et al. [**becker2007b**]