Generation of Mediation Modules for Personalization in Conceptual Content Management Systems

Student Project
submitted by

Mariya Denysova

supervised by
Prof. Dr. Joachim W. Schmidt
Sebastian Bossung

Hamburg University of Science and Technology
Software Systems Institute (STS)

Declaration

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Managing complex web cites nowadays is a big challenge. Every user visiting a web cite would like to get on it the information fitting exactly his/her needs in a clear and well understood manner. Different users have different interests, so if a publisher provides the same information for all cite users, only a small target group of users would be interested in the information provided. To reach wider target audience publishers can make use of cite personalization, i.e., providing for different users, different information so that it reflects the userss context, their specific background and interest as well as the heterogeneous infrastructure to which the content is delivered and in which it is presented [Bol03].

This the understanding of personalization from the content delivery point view. But there is another perspective to personalization from the content creation point of view. This means that before content is delivered it has to be created. The process of content creation is a collaborative process, involving many people. Different people or groups of people work on the same content modifying, enriching it, contributing to its value and finally publishing it. At different times different people have a sort of personalized content, i.e., the one they are working currently with, different from content published on the site.

These two aspects of personalization have a common problem, which is the providing of a unified view both to the public information, visible to other users, and to personalized information, specific for different users. Users should not know which part of the information is public and which is private. To solve this problem a mediator should be introduced, which would combine these two types of information and produce one common view to it. The purpose of the current work is to solve mediation problem for the personalization in conceptual content management systems.

1.2 Related work

1.2.1 Personalization

The world-wide web today is the largest information source, which is still growing exponentially. On the other hand the individual's capacity to digest information is fixed. "The full economic potential of the web will not be realized unless enabling technologies are provided to facilitate access to web resources. Currently web personalization is the most promising approach to remedy this problem"[SBK03]. Web

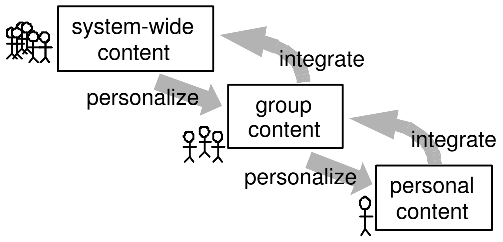


Figure 1.1: Personalization allows local change and reintegration of content [SBS05]

personalization is defined as any action that adapts the information or services provided by a Web site to the needs of a particular user or a set of users, taking advantage of the knowledge gained from the users navigational behavior and individual interests, in combination with the content and the structure of the Web site [EV03, Bol03]. This is the earliest and the most common interpretation of the notion "personalization". But nowadays almost every researcher understands this term differently. The meaning of personalization varies also depending on the kind of system it refers to:

Version Control, Community and Collaboration Management Systems

These kinds of information systems incorporate one common function: collaborative knowledge creation. "Knowledge creation" scenario is potentially even more important for applications, as knowledge can only be accessed after it has been created [KA03]. Personalization in this kind of systems is discussed in connection with three types of workflows [SBS05] (see figure 1.1):

- personalization of given content and structures
- sharing of personalized content amongst users
- making the views of users available to the general public.

In Version Control Systems these processes are traditionally called: *check out*, *submit* and *merge* accordingly.

Adaptive Information Systems

Generally the *adaptive system* is a system that can change its behavior according to the changing environment. Though some authors as Koch and Rossi consider adaptability only in connection with users' behavior [KR02]. They distinguish three different forms of adaptation according to the objects that may change during personalization: at content-level, at link-level and at presentation-level [KR02, RSG01]. "Personalization of software is actually nothing else than a particular case of software adaptability"[Bon03].

1.2.2 Conceptual Modeling

Conceptual modeling is a challenge during the last few decades. Idea of having means to model real world entities by means of computer science brings to the possibility to get the artificial intelligence. Generally *conceptual modeling* (also known as *information modeling*, *semantic modeling*, *knowledge representation*) "...is concerned with the construction of computer-based *symbol structures* and symbol structure manipulators which, according to mentalistic philosophy, are supposed to correspond to the conceptualizations of the world by human observers"[BMW82, BB03].

The object (objects) being modeled is usually called *domain* of application [Nii04, Wid98].

Abstraction mechanism is one of the tools of conceptual modeling used for organizing information. It has three constituents [BB03]:

- thinking of objects as wholes, not just a collection of their attributes/components ("aggregation");
- abstracting away the detailed differences between individuals, so that a class can represent the commonalities ("classification");
- abstracting the commonalities of several classes into a superclass ("generalization")

In order to express the result of conceptual modeling a *modeling language* is needed. According to [Nii04] a *modeling language* or a *formalism* is a well-defined technique (language with possibly some guidelines of how to apply it, and often some form of graphical visualization) "...used in modeling for expressing something about the domain of application". The broad discussion of existing modeling languages can be found in [Nii04].

Usually when speaking about modeling languages for some definite domain, researches use the term *domain specific languages(DSL). Domain specific languages* provide specialized language features that increase the abstractions level for a particular problem domain [CEG$^+$98]. Usually they are considered as programming languages for special domains, because they are used not only for modeling but also for processing by application generators (see 1.2.5) [Wid98, Bat03, SB00, BLS98].

The *symbol structures* resulting from conceptual modeling are often called *conceptual schemas* [Mar02a, Nii04], or sometimes *otologies.*

An ontology is a collection of concepts and their interrelationships, which provide an abstract view of an application domain [Kha00]. Still there is often a confusion about this notion [Gua98]: "In some cases, the term 'ontology' is just a fancy name denoting the result of familiar activities like conceptual analysis and domain modelling, carried out by means of standard methodologies. In many cases, however, so-called otologies present their own methodological and architectural peculiarities. On the methodological side, the main peculiarity is the adoption of a highly *interdisciplinary approach*, where philosophy and linguistics play a fundamental role in analyzing the structure of a given reality at a high level of generality and in formulating a clear and rigorous vocabulary. On the architectural side, the most interesting aspect is the *centrality* of the role that an ontology can play in an information system, leading to the perspective of *ontology-driven* information systems."

1.2.3 Content Management Systems

The term "Content management" is relatively new and is used to refer to many different things. Dictionaries don't provide any good insight to this notion. There are hundreds of different definitions of it. The broad discussion on the topic is given in [Gil00]. Without going into details we can conclude that broadly speaking, content management describes a process that allows people to more easily create and update content, usually on websites but not only. Gerry McGovern says [McG04]: "Content management is about getting the right content to the right person at the right time at the right cost".

What is "Content"?

According to [Boi04] content, stated as simply as possible, is information put to use, i.e., when it is packaged and presented (published) for a specific purpose. More often than not, content is not a single piece of information, but a conglomeration of pieces of information put together to form a cohesive whole.

The researches distinguish the three notions: *information*, *data* and *content*. *Information* is usually what people deal with in their ordinary lives. Usually it is semi- or non-structured, different in its meaning and importance. Computers cannot process information directly. Information is complex, and rife with relationships that are important to its meaning but impossible for a computer to decipher. To be processed by computers information should be turned to data. *Data* is simple, and all its relationships are clearly known (or else ignored). Data is always structured, similar data chunks are of similar importance and of similar meaning. The major steps made in the direction of artificial intelligence are the steps on the way of making computers work not with the data but directly with the information. The introduction of the notion "content" is such a step too. *Content* is a compromise between the usefulness of data and the richness of information. Content is rich information that you wrap in simple data. The data that surround the information (metadata) is a simplified version of the context and meaning of the information [Boi04].

Because content is an intermediate step between information understood by humans and data understood by computers content has both features of information and data. The properties common with information are: indirect important relations between different content parts, and the influence of domain semantics on the meaning of content. The property common with data is the representation of content as manageable reusable chunks, stored and processed by computers.

The main idea behind the notion "content" is the enrichment of data with *metadata*. *Metadata* is what brings additional value to the content itself and what gives to the content the features of information. Metadata may store all kinds of additional data about the data itself. This can be some domain specific data, directly inputted by the user of CMS who creates content or otherwise gathered somehow indirectly, or the data about the presentation of content. The ways of organizing metadata, storing it, the ways of gathering it and decisions about which metadata to gather vary significantly in different implementation of different CMS.

Another idea exploited in CMSs is the desire to combine and reuse content. Combining and reusing is a trivial task when dealing with data, and it becomes quite a challenge when speaking about content. To be able to reuse content it is convenient to divide it into manageable chunks called *components* (*content components*) [Boi04, Tri05]. *Content Components* divide information into convenient and manageable chunks. They "...are a set of discrete objects whose creation, maintenance, and distribution can be automated. They typically share some common attributes, such as format or length, and they should be self-contained, not needing the context of other components to be meaningful" [Boi04].

The term "component" will be also introduced with respect to the *software components* in the section 1.2.3. In that section the definition of *software components* use similar notions: "self-contained" pieces, "interaction with other components", "reusability". This brings to the possibility to implement the information components through software components, and introducing one-to-one correspondence between *information components* and *software components*, e.g. as implemented in DRACO technology [Nei89, Nei01, dPLSdF94]

Content Management Systems

A Content Management System (CMS) is not really a product or technology but a catch-all term that covers a wide set of processes that will underpin the "Next Generation" large-scale web site [BL01]. At first glance, content management may seem a way to create large Web sites, but on closer examination, it is in fact an overall process for collecting, managing, and publishing content to any outlet, as the following list describes [Boi04]:

In *collection*: Creation or gathering information from an existing source, possible processing and conversion to a proper format, and final aggregation the CMS by editing, segmenting into chunks (or components), and adding appropriate metadata.

In *management*: Creation of a repository that consists of database records and/or files containing content components and administrative data (data on the system's users, for example).

In *publishing*: Making content available by extracting components out of the repository and constructing targeted publications such as Web sites, printable documents, and e-mail newsletters. The publications consist of appropriately arranged components, functionality, standard surrounding information, and navigation.

1.2.4 Component Software Architecture

Software Components are seen nowadays as the main instrument to facilitate software reuse. There are many definitions of the term "component", such as a component is "a self-contained functionality or application which can interact with other components"[KBA00].

Usually researches also introduce the notions of "interface" and "realization" to the definition of component, saying that a component always consist of the *interface* visible externally outside of the component and the rest part, called *implementation*[BO92, PDH99].

Anyway this definition doesn't show the difference between "modules" and "components", but "'component' is a type of 'module', sharing some generic properties with 'modules' that are not 'components"' [PDH99]. The main feature distinguishing *modules* from *components* is that components are meant to be reused in different applications [Nei92, KBA00, Nei80, BO92]. The possibility to reuse components on one hand brings to the creation of component libraries, from which the developers can choose the components for their applications, and on the other hand triggers researches in the direction "easy" or even *dynamic* replaceable components in one application.

The problem of dynamic replacement is discussed in [KBA00, Blo83] as the possibility "to allow the programmer to add or remove components from the application at run-time. Such decision could be influenced both by necessity and the changing performance characteristics of the application".

Batory [BO92] discusses the problem of component exchangeability. He introduces the notion of "realm" as a set of components implementing the same interface and the notion of a "symmetric" component, as the component that makes calls to other components of the same realm, i.e., the components that implement the same interface as the caller. Batory claims in his work that "the true building blocks for some realms are symmetric components" and that "not recognizing such components is a lost opportunity for achieving reuse on a large scale". Such components are often called *plug-compatible*, *interchangeable.*

Neighbors, the author of the DRACO technology [Nei89, Nei01, dPLSdF94], in his work [Nei92] discusses the approaches to organizing an easy in use component library.

He comes to the conclusion, that using specialized domain languages is an alternative to using program libraries. “A library would not have been as successful because the burden of using the library and knowing the interconnection limitations is placed upon every potential user of the library. Having a domain-specific language that ties the library together removes this burden at the expense of learning the language.”

1.2.5 Program Generators

A large group of computer scientists came to the same conclusion as Neighbors and “...developed a new subarea of Software Architectures, that is *software generators* (also *application generators*)” [Bat97]. In the technical sense, *application generators* are compilers for *domain-specific programming languages* (DSLs) [SB00], or the realizations of domain models that explain how software systems in a target domain can be assembled from previously written components [BDG$^+$95]. The process of modeling families of software systems by software entities such that, given a particular requirements specification, a highly customized and optimized instance of that family can be automatically manufactured on demand from elementary, reusable implementation components is usually called *Generative Programming* [CEG$^+$98, Bat03, Bat04]. These requirements specification are usually written using domain-specific languages.

Batory in one of his works [SB00] claims that one of the reasons for using generators is that “the specification languages that generators implement (domain-specific languages) are much more concise and convenient than the language of the produced program (called the *target language*), the translation of specifications to target code is done correctly and quickly, thereby substantially increasing programmer productivity ” [SB00]. In [Bat97] he also says that “the basic distinction between research on generators and software architectures is that the components that generators compose to construct systems are designed to be plug-compatible, interchangeable, and interpretable ” (as defined in section 1.2.4).

Scientists distinguish two types of generators: self-sufficient, *stand-alone translators* (in much the same way as compilers for general-purpose languages) and *program transformation systems* (*transformation generators*) [SB00]. A successful example of a system that builds applications out of domain specific languages using a set of *transformation generators* is DRACO system [Nei89, Nei01, dPLSdF94].

According to Batory [SB00] application generators have the standard internal form of a compiler with a front-end, translation engine, and back-end component. The front-end is responsible for the one-to-one mapping of the input form to an equivalent but more convenient internal representation. Typical input specifications are in text format, in which case the front-end consists of a conventional lexical analyzer and a parser. The translation engine implements transformations on the intermediate representation. Usually transformations are expected to satisfy some correctness property: the transformed program should have the same semantics as the original, if not for all inputs, at least under well-defined input conditions. Translation engines and transformations are the core of generators and are discussed in detail in the next section. The result of applying transformations to the intermediate representation is a concrete executable program. The concrete program, however, is still represented as a flow graph or an abstract syntax tree. Mapping from the intermediate representation to program text is straightforward and is the role of a generators back-end. Generated code is usually in a high-level programming language.

The program code the purpose of which is to generate some other code is usually called *metacode* [CEG$^+$98].

Generators are often considered as an integral part of an information system itself. Thus, when speaking about component architecture of the system some scientist, like e.g. James M. Neighbors and others, distinguish two types of components: *compositional* components, that encapsulate the code that applications execute at run-time and *transformational* components , that encapsulate algorithms that generate the code that applications execute at run-time [JPB97, Bat97]. Thus, according to the definition above this components contain *metacode*. Transformational components generate compositional components that are specifically optimized and customized to a particular application [Bat97].

1.2.6 Mediators

The term "mediator" was originally introduced by Gio Wiederhold as "... special kind of components or modules, that exploit encoded knowledge about some sets or subsets of data to create information for a higher layer of applications"[Wie92]. According to Wiederhold [Wie92] the main role of mediators to *mediate* between the other modules, such as for example data resources. These modules perform administrative role and gain technical knowledge from underlined modules. The software which mediates is common today, but the structure, the interfaces, and implementations vary greatly. Wiederhold also mentions that one possible usage of mediators can be the merging of information from multiple databases and providing independence from data resources, as well as providing access to other mediators.

1.2.7 Conceptual Content Management Systems

Conceptual content management systems refer to a special kind of content management systems, which manage content through the management of assets (figure 1.2). "Assets are ontological descriptions used to classify content"[Seh03].

The purpose of Content Management Systems is to organize a complex mixture of media content and to present this mixture trough domain specific conceptual models [SS04].

Conceptual content management systems are supposed to be used in collaborative environments, such as research and learning systems, to support user activities such as acquisition and exploration of concepts as well as creation, enrichment, publication, and sharing of content [SBS05]. The example of such system working successfully for several years is GKNS project [1].

The architecture of conceptual content management systems is a component architecture. Some of its components are *transformational* components as defined in section 1.2.5. This means that some of the components represent an application generator.

If fact the whole architectural principles of conceptual content management systems are quite similar to Neighbour's DRACO system but applied to a different domain, namely content management domain, and using not transformational but a stand-alone application generator. The more detailed discussion on this topic will be given further.

1.3 Structure of the thesis

The first chapter was devoted to the overview of the latest progress in the fields tightly connected with the topic of the current thesis.

In chapter two I will formulate the task to my student project and will describe the architecture and workflow scenarios for the conceptual content management systems,

[1]GKNS web site http://www.wel.de/gkns

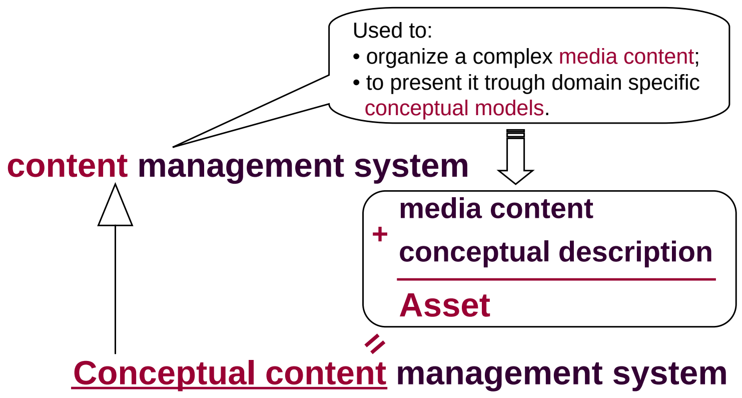

Used to:
• organize a complex media content;
• to present it trough domain specific conceptual models.
content management system
media content
+
conceptual description
Asset
Conceptual content management system

Figure 1.2: Conceptual Content Management Systems

get the reader acquainted with all the notions necessary for the understanding of the rest of the work.

In the third chapter I will outline the requirement to the generator that was developed during the project.

In chapter four I will provide the design solutions.

Chapter five will be dedicated to the implementation problems.

In chapter six I will make my conclusions about the work done and outline the directions for the future work.

Chapter 2

Background

2.1 The Problem

In different kinds of applications, which use conceptual content management systems, a user needs to have a possibility to personalize the content. Personalization usually implies adaptation and customization of data content or the representation of this data.

We can distinguish between asset instance personalization and asset schema personalization (more in section 2.2.7). The scope of the project deals only the first way of personalization (which is the easier case). Asset instance personalization implies that user is able to change only the values of asset characteristics and not the asset schema.

The difficulty of using the personalized components is that whenever a personalized component exists it exists along with an analogous public component. Whenever a request to a component is done, a system has to decide to which one of these two components it is: the personalized or to the public one. To perform this task a special mediation module is required. The task of the current work is to develop a generator that generat application scecific mediation modules.

2.1.1 Example

A group of art historians publish a painting called "Mona Lisa" in the conceptual content management system. The notation used here are the *Asset Definition Language (ADL)* and *Asset Manipulation Language (AML)* languages described further in sections 2.2.1 and 2.2.2.

```
class Picture {
    content contents: Image
    concept characteristic title : String
            relationship painter : Artist
}

monalisa = create Picture {
    contents := de.tuhh.sts.wel.Media.MONALISA
    title := "Mona Lisa"
    painter := "Leonardo_da_Vinci"
}
```

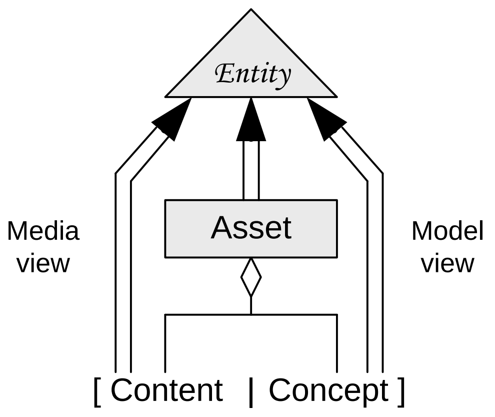


Figure 2.1: Assets represent entities by [content — concept] - pairs [Seh03]

Another group of artists, while studiying the works of Leonardo da Vinci is used to calling this painting "Jokonda" and wants to work with this name of the painting.

The possible solution to this. The second group of art historians pesonalizes the instance of class Picture called "Mona Lisa" and change the value of characteristic called title for "Jokonda":

```
jokonda = modify monalisa {
    title := "Jokonda"
}
```

2.2 Conceptual Content Management Systems

2.2.1 Assets

As was said in section 1.2.7 assets are "...ontological descriptions used to classify content". This means that assets serve two roles:

- give conceptual description of domain entities - concept part;
- incorporate content part of domain entities - content part.

Thus assets represent intimately allied content-concept pairs which represent and signify application entities [SS03] see figure 2.1.

Assets can inherit from each other thus forming ontological hierarchies which are called asset models.

M.L. Brodie [Bro84] defines tools associated with data models as "languages for defining, manipulating, querying, and supporting the evolution of databases". These languages are *Data Definition Language (DDL)*, *Data Manipulation Language (DML)*, and *Query Language (QL)*. The same approached is used for asset models.

The following languages are defined to manage asset models:

Asset Definition Language (ADL) used to define asset schema;

Asset Manipulation and Query Language (AML) used to manage asset instances.

2.2.2 Asset Definition Language

To support expressiveness of asset modeling the concept part has tree constituents [SS04]:

- characteristic values,
- relationships between assets, and
- rules (types, constraints, ...)

Asset definition language is a schema language. Users can describe domain schema using this language and associate content with conceptual descriptions. According to section 1.2.2 asset language can be considered as a *domain specific language*, describing the domain of content management and used as input schema for an *application generator*.

Asset definition language is a class-based language and is used to define asset classes [SS04].

```
class Picture {
    content contents: Image
    concept characteristic title : String
            relationship painter : Artist
}
```

The section `content` references the content part of asset definitions using content handels. E.g. it can be a URL address where the resource is stored, or a file name in the file system. The colon mark divides the name of the content resource and its type.

The section `concept` is used for description of the concept part of an asset.

The key words `characteristic`, `relationship`, `constraint` describe characteristics, relationships and constraints respectively. Colon marks separate their names from their types.

The inheritance relationships between asset classes are expressed using the key word `refines`:

```
class Portrait refines Picture {
    concept relationship portraitOf : Person
}
```

A set of asset classes describing entities of the same domain is called *asset model*.

```
Model Paintings
class Picture {
    ...
}
class Portrait {
    ...
}
```

Different asset models representing different domains can be imported to other models modeling larger domains.

For more details about asset definition language see [SS04].

So asset definition language complies good with the *abstraction mechanism* introduces in section 1.2.2.

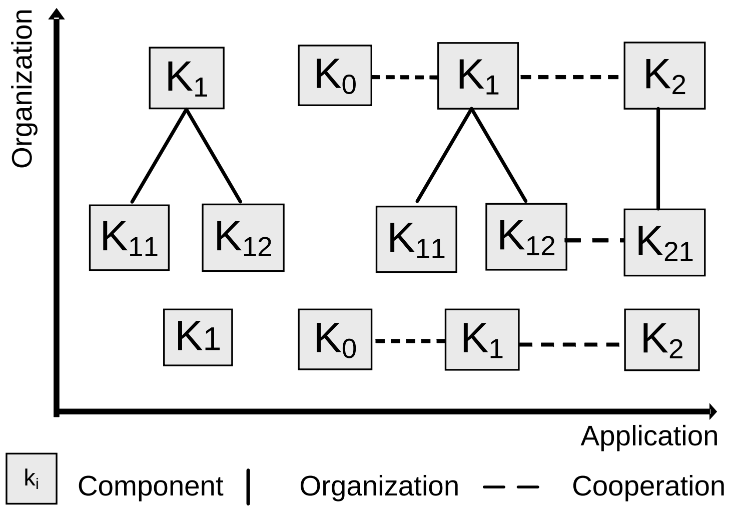


Figure 2.2: Component Configuration [Seh03]

2.2.3 Module-Component Architecture of Conceptual Content Management Systems

As was stated before conceptual content management systems are open and dynamic adaptive computer systems. This is possible due to some architectural features of conceptual content management systems.

The architecture of the conceptual content management systems is a component architecture. The combination of such components represents the combination of coexisting domains (sub models). Thus, in contrast to Neighbours approach (e.g. in [Nei92] or DRACO-approach [Nei89, Nei01, dPLSdF94]) of using "one software component for each object or operation in the domain", here one component for each domain is created.

A component represents assets which describe a domain in a definite context. This means that we will need more than one component to represent [Seh03]:

- assets from different domains (several assets from one domain, i.e., one model, goes to the same component);
- assets describing the same domain but from different points of view (in different contexts).

Therefore we have two types of relationships between components: *cooperation* (communication along usage structure) and *personalization* (individualization along organization structure). The context organization of users, usually represented as project groups, brings to the corresponding organization structure of the components [Seh03] (see figure 2.2).

The components in their turn consist of modules. In fact "modules", as they are called in conceptual content management systems, are real *components* in the sense as discussed in section 1.2.4. The term "module" is used for convenience purpose. More over they are *symmetric* components (see section 1.2.4), thus can be dynamically replaced. This architectural property makes for *system dynamics* (see section 2.2.7).

One can say that there are two kinds of components in conceptual content management systems that provide two different forms of reuse [Seh03]:

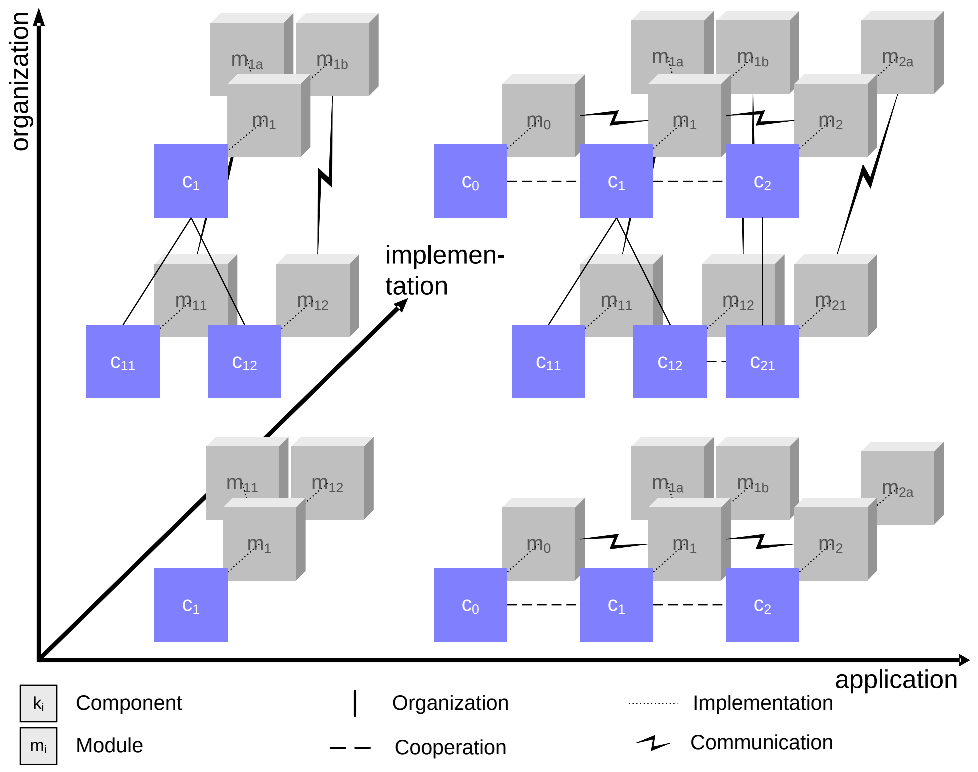


Figure 2.3: Components Implementation through Modules [Seh03]

- components (called "components") contribute to assets reuse and
- components (called "modules") make for functionality reuse.

These two kinds of reuse are an example of *separation of concerns* approach according to [TO01, TOHS99, Mar02b] and others.

An example of relationships among modules and components is shown on figure 2.3.

2.2.4 Modules

There are several types of modules to realize several functionalities. Each component can be built of any number of such modules, and be dynamically rebuild when needed.

The short description of the types of modules and their functionality is as follows:

client modules used to access standard components managing the asset' content and data; the only modules that can store persistent information;

transformation modules used to adjust schemata, thus allowing modules generated from different ADL schema revisions communicate;

distribution modules allow the incorporation of modules residing on different networked computers;

mediation modules to glue the modules of a conceptual content management system together by delegating calls to other modules and combining their responses in different ways;

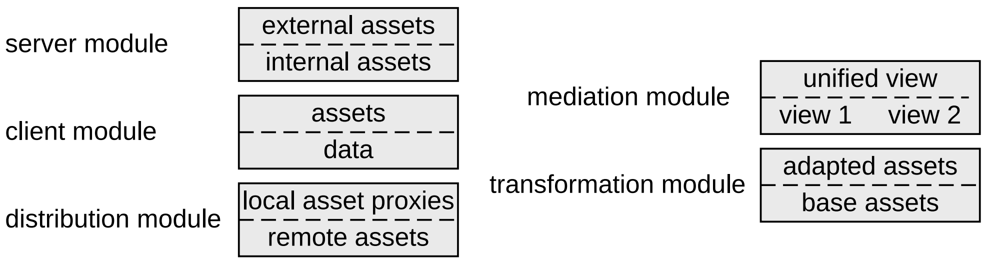


Figure 2.4: Module Kinds [Seh03]

server modules offer the services of a conceptual content management system following a standard protocol for use by third party systems.

Client modules are bottom-layer modules, the other modules are on top of them and delegate calls to client modules. Thus, the architecture of a component is a typical layered architecture.

The types of components and their typical configuration within a component is shown in figure 2.4.

As was already mentioned all components are *symmetric* and implement the same interface, called `Module`.

2.2.5 Conceptual Content Management System Generation Scenario

It was said in section 1.2.7 that some components of conceptual content management systems are *transformational* components used to generate application specific code. Before discussing the principles of the application generator framework within conceptual content management systems I will give an overview of the whole generation scenario, which is depicted in figure 2.5.

To generate a conceptual content management system for some assets a user writes asset definitions in the asset definition language. Then this asset schema is processed by the *compiler framework*, which is, actually, the application generator. The output of the application generator are files in a high-level programming language, namely Java, representing components and modules conforming to the above described architectural principles.

If a user wants to create, modify or delete asset instances, he can either use *asset query and manipulation language (AML)* or use a client code making calls to the interface of the generated modules.

Once a user wants to change asset schemata, the whole procedure is repeated.

2.2.6 Compiler Framework

The Compiler Framework is nothing else but a stand-alone application generator as defined in section 1.2.5.

The compilation scenario follows the compilation process in Model Driven Architectures (MDA). The asset compiler creates a platform independent model from a domain model. The platform independent model is then translated into a running software system [SS04].

A compiler consists of frontend and backend. The frontend lexes and parses asset definitions resulting in an intermediate model. The backend contains API and Module

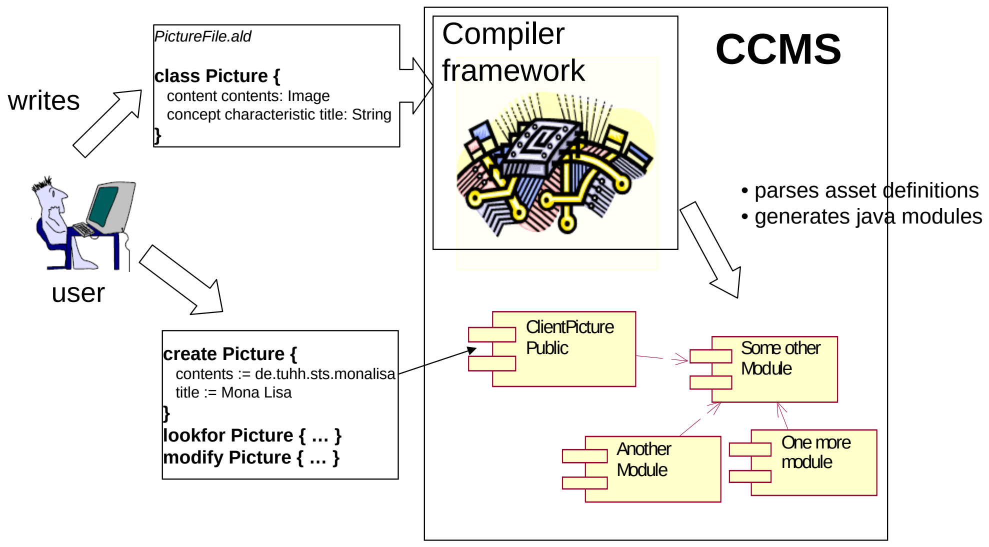


Figure 2.5: Generation Scenario

generators that generate code out of the intermediate model. Each generator can get several symbol tables as its input, and produce exactly one symbol table as its output (see figure 2.6). A *symbol table* contains the object representation of the generator output and provides methods to read this information by other generators at runtime.

API generator generates a set of interfaces, common for all modules of the same component and stores them in *API Symbol Table*. To be able to generate the implementation of a module the generator has to have information about API of this module. It gets this infomation from the symbol table resulting from the work of API generator.

Usually each module generator generates implementation for one module. Though often to generate the implementation for a module more than one generator is needed. Such generators produce some part of module implementation and communicate with each other using symbol tables. This is a data-driven communication similar to the one used in “pipes and filters” architectures [SG96].

2.2.7 Personalization

Personalization as understood in connection with conceptual content management systems “is the ability of a system to adapt to the individual needs of each user” [SBS05]. This definition of personalization is broader than in most contemporary works, because it not only implies that different users perceive different information from the system, but also that they have possibility to modify a system through personalization. This brings to the constant evolution of the systems with the evolution of information content it manages. Thus the system gains the ability to transform itself without additional reprogramming. This is possible due to two properties of conceptual content management systems [SS03, SBS05]:

schema openness: users can change the schema on-the-fly adapting their asset models according to the requirements of the entity at hand;

system dynamics: the implementation of the system changes dynamically, following any on-the-fly schema modifications in the running system.

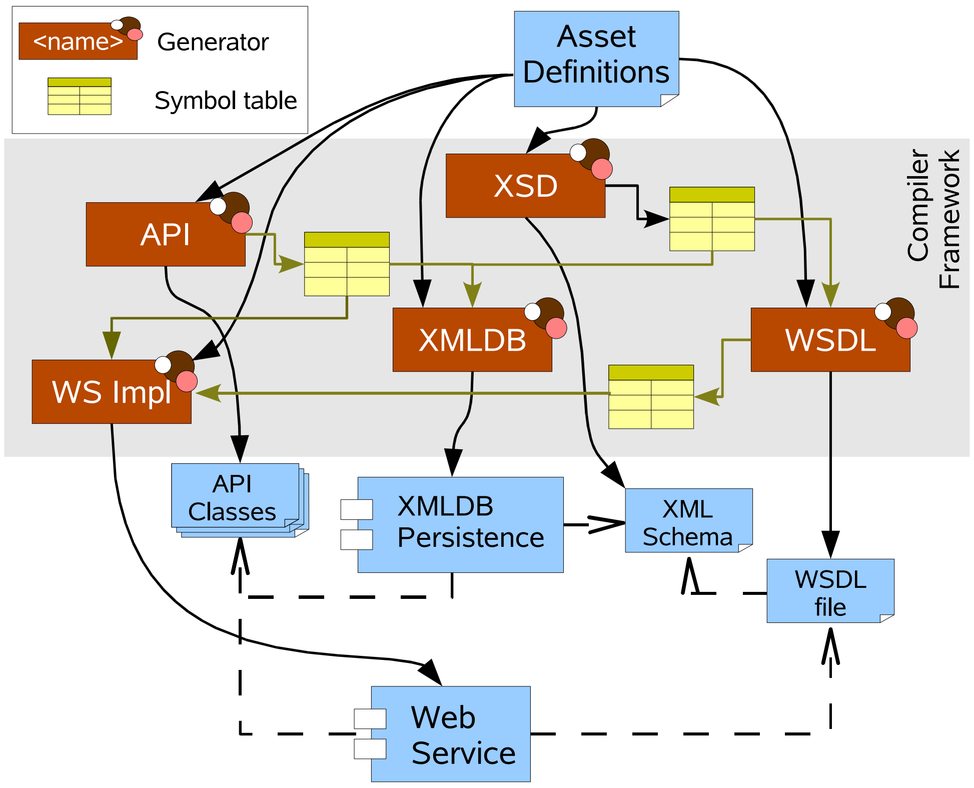


Figure 2.6: Compiler Framework

The initial system configuration is built according to the schema provided by the user. If, for any reason, a user of a system wants to change asset definitions and thus the system configuration too, there exist two scenarios to do so:

"easy" A user just changes the assets to fit the new needs. The system modifies automatically due to dynamics property.

"legal" A user personalizes assets, changes them and then publishes making changes visible to other users and resolving conflicts. The systems modifies according to this actions.

The problem with the first scenario arises, because the content already existed in the content management system is described by the old schema, new changes cannot be applied to it, nevertheless it should be kept further in the system, it can't be just thrown from it. On the other hand the other users of the system may not agree with the new changes both to asset schema and asset instances, and may want to continue working with the former version.

The two processes of the second scenario, personalization and publishing, deal with these problems.

According to other taxonomy [Seh03] there are tree grades of personalization:

- Asset creation
- Establishment of relationship between public and personal Assets

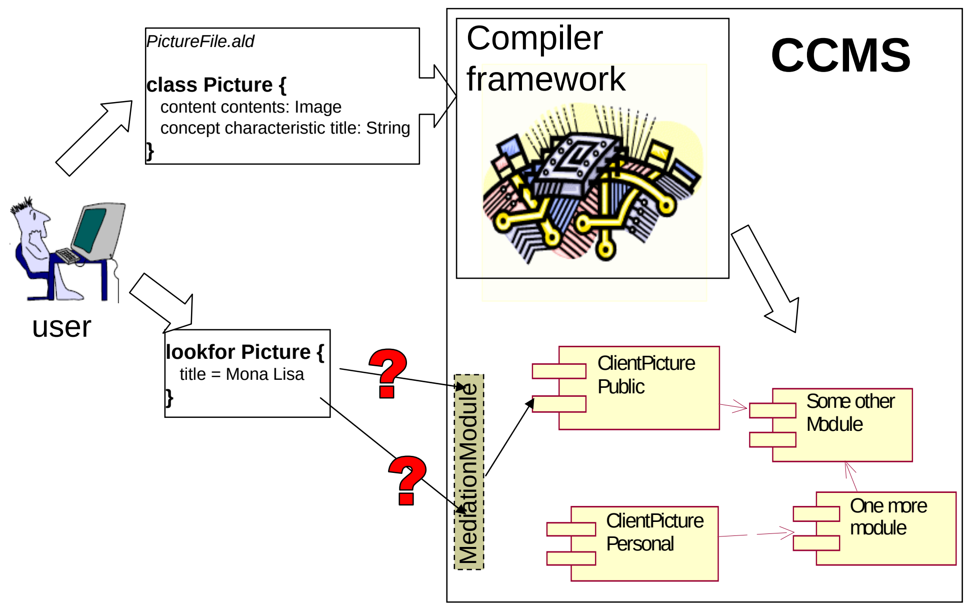


Figure 2.7: Mediation Modules for Personalization

- Transformation of public asset to personal view

These three forms of personalization, in the order from 1 to 3, require constantly increasing content management system support.

Personalization happens on two levels:

asset instance personalization (content personalization) asset instance changes (e.g. change of a characteristic value, addition of deletion of relationship), asset schema remains the same;

asset schema personalization (structure personalization) changing asset definition (e.g. addition or deletion of characteristics).

Asset instance personalization cause the creation of additional component, *personalized component* and generation of a mediation module to delegate calls either to public or to personalized (private) component (see figure 2.7).

The structure personalization will also require an additional transformation module to adjust different schemas of public and private components. This is a more difficult case and is not considered in the current work.

Chapter 3

Requirements

3.1 Requirements to the Generator

The non-functional requirements to the software developed within the scope of this work include in the following:

1. The developed software must be a module generator of the conceptual content management system compiler framework back-end to make use of API Generator.
2. The generator must be developed using the Java Code Generation Toolkit (see section 5.2), which provides a convenient form to keep Java code in object form and then print it in text files as well as store in the *generator symbol table.*
3. The generator must generate Java code for mediation modules, that delegate calls either to the public or to the private component to provide a unified view on these two base modules and make them look from outside like one module. The requirements to the mediation modules functionality is described below.

3.2 Functional Requirements to the Mediation Modules

The functionality of the mediation modules in general is expressed on the Use Case diagram in figure 3.1.

As shown on the diagram a user of a mediation module can perform such activities as *asset creation*, *asset modification*, *asset deletion*, *asset retrieval.* The last activity asset publishing should also be available for a user, but only first four are within the scope of the current work.

By the *User* in this diagram I mean any other system module, which is above the mediation module in the layered architecture of the component, or any external user of the system (e.g. external client software), if the mediation module is the topmost module in this architecture.

The functionality of each use case is described below on collaboration diagrams.

3.2.1 Asset Creation

The diagram for asset creation scenario is shown in figure 3.2.

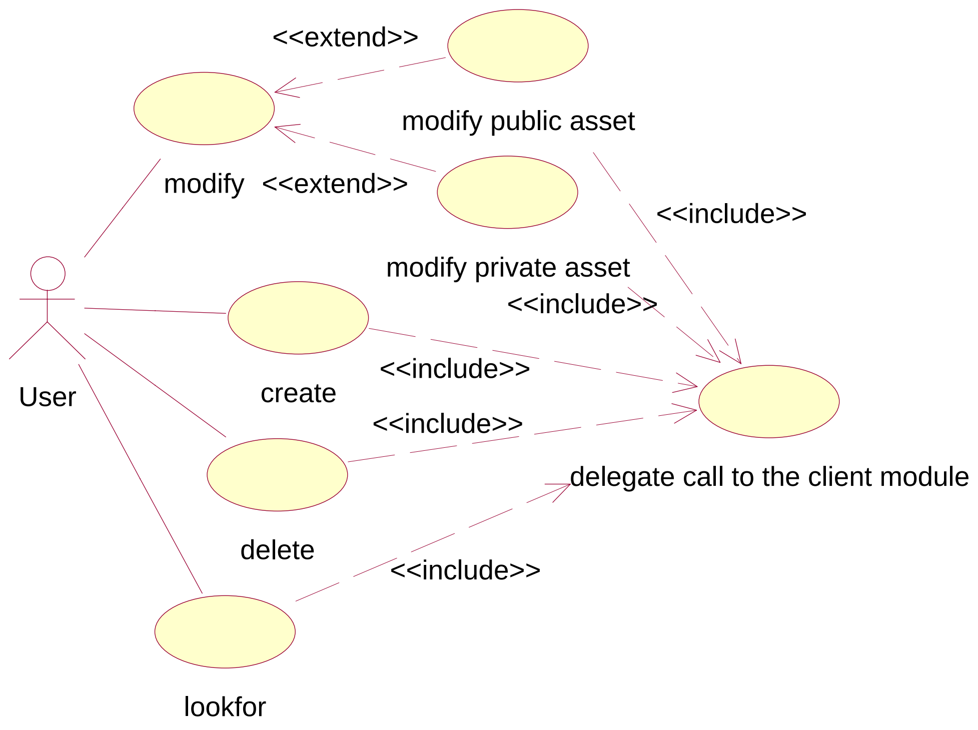


Figure 3.1: Mediation Modules Use Case Diagram

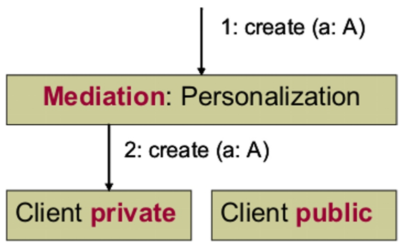


Figure 3.2: Asset Creation

To conform to the personalization workflow as described in section 2.2.7 the new assets are always created in private modules and reside there until they are published.

If no private component for the given asset model exist, then by creation of new asset for this model and new private component should be generated. If the private component for that module exist then the new asset would be created in that component.

3.2.2 Asset Retrieval

Asset retrieval corresponds to the `lookfor` operation of asset query language. One should be able to retrieve assets which satisfy the constrains of this operation.

An external user of the system should not see the difference between public and personalized assets. On the query both assets from public and private module meeting the query constraints have to be retrieved. In case of existence of both public and personalized variants of the same asset the personalized instance should be returned. The mediation module has to perform this task of searching in both modules and merging the results afterwards. The way to do that is shown in figure 3.3.

The merge operation in this scenario produces a set of assets out of two other sets and is performed as follows:

1. all instances from the first set (personalized) are added to the result set;
2. for each instance of the second set (public) the check is performed, checking if there is the instance in the first set, that corresponds to the instance in the second. If no, then the instance of the second set is added to the result set, else it is not added.

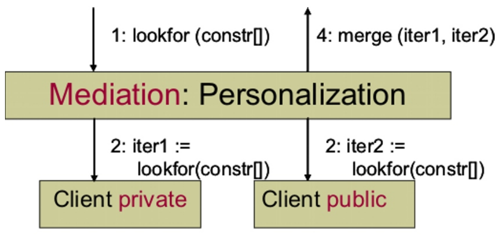

1: lookfor (constr[])
4: merge (iter1, iter2)
Mediation: Personalization
2: iter1 := lookfor(constr[])
2: iter2 := lookfor(constr[])
Client private
Client public

Figure 3.3: Asset Retrieval

3.2.3 Asset Modification

The problem with asset modification is that when a user wants to modify an asset, generally he doesn't know if this asset is public or personalized, i.e., he doesn't know if it resides in the public or personalized module and doesn't know in which of them to search. He even does not know that there actually are two different components to store assets. In this situation the mediation module is needed.

The mediation module gets the call from its user to modify an asset. At this call it first tries to retrieve this asset from the private module, thus checking if the asset was already personalized. If the retrieval succeeds and the needed asset is retrieved, the mediation module calls the modify operation of the private module for the retrieved asset. Otherwise, if the needed asset was not found in the private module, this means that no private variant of the asset exist. The modification operation in this case is nothing else but a personalization of a public asset. In this case a copy of the public asset is created in the private module and then this copy is modified according to the initial request.

Thus there are two different scenarios for asset modification: the case of modification of public asset and the case of modification of private asset, i.e., the asset that was already personalized before, but was not published by that time. These two cases are shown on the figures 3.4 and 3.5 correspondingly.

3.2.4 Asset Deletion

The question about asset deletion is currently open in the theoretical works on conceptual content management systems. The question namely is: should the published asset be deleted?

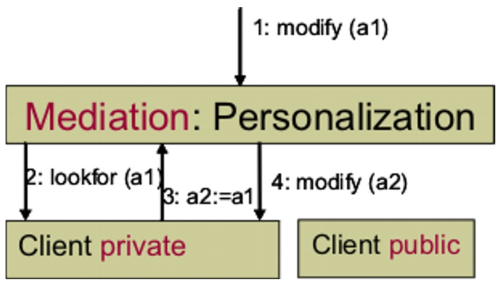


Figure 3.4: Personalized Asset Modification

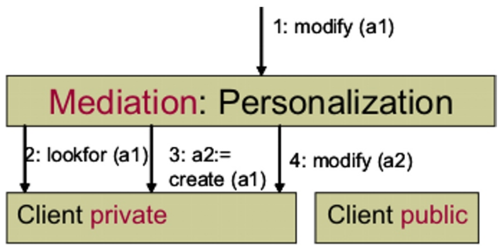


Figure 3.5: Public Asset Modification

The current answer to this question is: No. Because the asset is a part of the real world being modeled and it cannot be just deleted.

In this work we will allow the deletion of private assets and forbid the deletion of public ones.

The deletion scenario is a simple one. The assets meeting the constraints are retrieved from the private module by `lookfor` operation and then deleted.

Chapter 4

The Design of the Mediation Modules

This chapter provides explanations about the design of the *Mediation Modules*. Some parts of the *Mediation Modules* are non-generic and depend of the definitions of asset classes, thus, we need to base our explanation on a definite asset schema. The explanations are based on the following asset definitions:

```
model personalization

class Person {
concept
    characteristic name: String
    relationship origin1 : Person
}

class Student refines Person {
concept
    characteristic matrikul: String
}
```

4.1 Origin Linking

The requirements (see chapter 3) demand the mediation modules store the connection between public and private components. This connection must be stored persistently, because the personalized version of asset instances could be stored in the system for a long time before they are published, so this connection should survive after system restarts. But according to conceptual content management system architecture only client modules can store persistent information, the other modules are stateless.

The client modules are the only type of components that have access to the database, thus they can store the connection in some form in the database. But the only form, in which the client modules can store something in the database, is the the form of assets. All assets stored persistently are have to be described by the ADL schema. Here we come to the conclusion, that if we want to store persistently the connection between public and personalized assets, we have to explicitly introduce this connection to the ADL schema.

I considered two possible schema modification:

1. Modification of each asset class definition by introducing additional relationship `origin` storing the relationship to the public instance of each personalized asset or `null` for not personalized assets:

   ```
   a = class A { ... }

   ap = class AP{
       ...
       relationship origin : A = a
   }
   ```

   This variant is also suggested in [Seh03].

2. Introducing an additional asset class for each asset class defined by the user to store the relationships to public and to personalized instances:

   ```
   a = class A { ... }
   ap = class A{ ... }

   PersonalizationA{
       relationship public: A = a
       relationship private: A = ap
   }
   ```

We chose the first variant for the future implementation, because of several reasons:

1. It seems to be more natural.

2. From performance consideration.

   Given a personalized instance, to find its public instance only one `lookfor` operation is needed (namely lookfor ID, which is usually considerably faster then lookfor operations with other query constrains). The same holds in the other direction, i.e., given the public instance, to find the corresponding personalized only one `lookfor` operation constraining the origin field, whose value has to be the given public instance. This operation is slightly slower than the look for ID operation. On the other hand with the second variant first the search of associative entity is needed, with the constraints on public (private) relationship to find the proper associative entity storing the IDs of the needed private and public instances. Then one more search operation is needed, searching for ID of private (public) instance.

   The performance of lookfor operations differ considerably from one to another database implementation, however it is quite a time consuming operation.

But the first variant has one substantial drawback comparing to the second one, because the first variant implies schema modification of the user defined assets. This means that the new relationship `origin` will be visible for the user, and the user would be able to change this relationship, which he is not supposed to be allowed to do. How to tackle this difficulty is explained as the ideas for the future work in chapter 6.

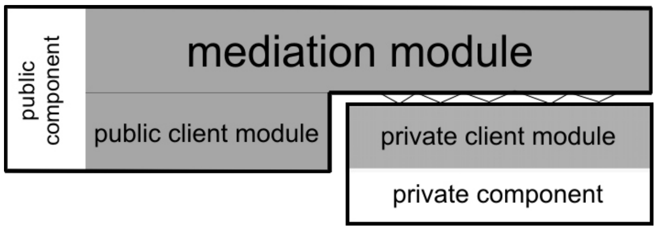


Figure 4.1: Module Configuration for Personalization

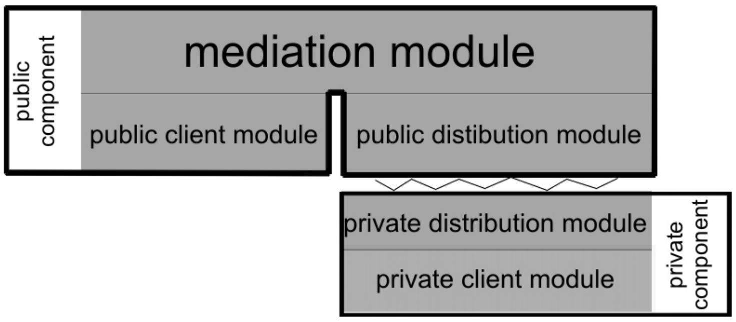


Figure 4.2: Module Configuration for Personalization

4.2 Component Configuration

As was stated in section 2.2.3 and is claimed in [Seh03] "assets describing the same domain but from different points of view (in different contexts)" should reside in different components. These words refer to the case of personalization, because the personalization is nothing else but description of the same domain from the point of view of different user groups. Thus in our case we should have two components: public and private one, and mediation module gluing them together.

One possible configuration is shown in figures 4.1 and A.1.

The more difficult case is when public and private components should reside on different computers. In this case an additional module has to be inserted into each of the components to glue them together by facilitating their communication over the network, using standard protocols. These two modules are called distribution modules. Fore more details on the functionality of distribution modules see [Seh03, SS03]. In this case the configuration will look as shown in figure 4.2.

These two variants of component configuration describe the situation from the "How should be?" perspective. Unfortunately the real situation is not so good. The current project was a simplified component configuration, that reduced significantly the problems with configuring components without loosing the full functionality of mediation modules.

For the current project the both public and personals modules are configured to reside in the same component. Thus, the configuration "as it is" is shown in figures 4.3 and A.3.

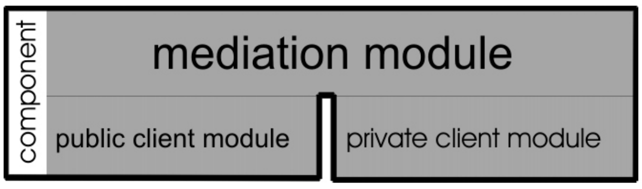


Figure 4.3: Module Configuration for Personalization

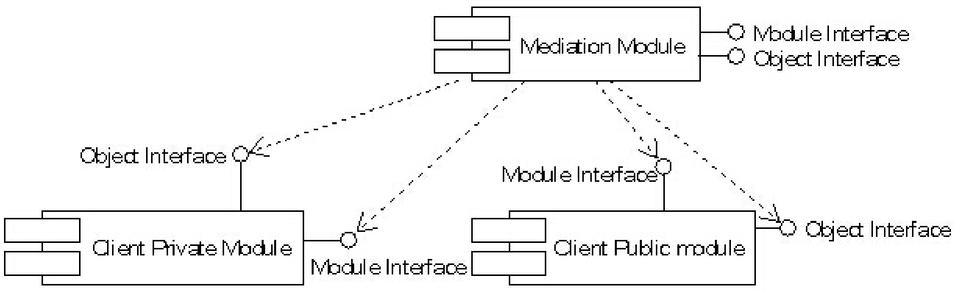


Figure 4.4: Module Configuration for Personalization

4.3 Interface Specification

API generator generates a set of interfaces. This set of interfaces is usually called *Object Interface*. One more interface, called *Module Interface*, is generic, i.e., not generated by the API generator. These are the interfaces defining the "legal" way for communication from outside of the system. They must be implemented by all modules in order they be *interchangeable* as defined in 1.2.4.

These interfaces are:

Module Interface One per asset model, i.e., one per component. A generic interface, i.e its name and the names of its methods don't depend on the names of asset classes names in the asset schema file.

Object Interface Consist of several non-generic interfaces, the name of the interfaces and their methods depend either on the name of asset model (in case one interface per asset model is generated) or on the name of asset class (if one interface per asset class is created).

Thus the component diagram, representing the mediation and two client module will look as show in figure 4.4.

This set of interfaces is prescribed by the architecture of the conceptual content management systems. More on the specification of these interfaces is written in [Seh03] and [SS03]. The need for two different interfaces `Module` and `Object` arise from two different approaches to building client applications. `Module` interface is useful to build generic clients able to work with different classes of assets, therefore clients have not be substantially changed when asset class definitions change. On the other hand `Object` interface exploits the idea of type safety and provides more flexible instruments fore working with assets.

Almost every task that can be done using `Object Interface` can be done also using `Module Interface`, still `Object` interface is often more powerful. However, there is a task, the implementation of which is almost impossible with `Module Interface`. It is execution of complex queries on assets. Assets can be retrieved using `Module Interface`, but `Module Interface` provide no tool for execution of queries containing sub-queries or queries with conjunctions and disjunctions of complex constraints.

Additionally, when discussing module configuration one more thing should be mentioned about assets "residing" in each module. Each module has to know what asset classes it is responsible to manage, also each asset should know by which module it is managed. This is done with the help of a metaclass which implements the generic interface `AssetClass`. Thus modules and asset classes should be in bidirectional association relationship with each other. More about it is in the section 4.4.2.

More on the interfaces and their realization by mediation modules is discussed below.

4.4 Realization of the Interfaces

The whole set of interfaces that mediation modules are to implement along with their descriptions and the most important methods of these interfaces is shown in appendix in table B.1. The key issues about the design of the implementation classes are discussed in the following chapter.

4.4.1 Realization of Module Interface

`Module Interface` provides several methods for asset creation, several methods for asset modification, also retrieval and deletion. The several `create` methods differs in their parameters and return type. The same applies to `modify`, `lookfor` and `delete` methods. As an example several method signatures of these methods are provided here:

```
public Asset create(AssetClass ac, MemberInitialization[] mi);
public Asset create(AssetClass ac, AbstractAsset aa);
public AssetIterator create(AssetClass ac, AssetIterator ait);
public NewAsset delete(Asset asset);
public AssetIterator delete(AssetIterator assetIterator);
public Asset lookfor(ID id);
public AssetIterator lookfor(AssetClass ac, QueryConstraint[] qc);
public AssetIterator lookfor(AssetClass ac, AbstractAsset aa)
public AssetIterator lookfor(AssetClass ac, AssetIterator it);
public AssetIterator modify(AssetIterator ait,
                                        MemberInitialization[] mi);
public Asset modify(Asset as, AbstractAsset aa);
public Asset modify(Asset as, MemberInitialization[] mi);
```

As we see all these method signatures are generic. They don't depend on the definitions of asset classes. The functionality of these methods for mediation modules also doesn't depend on the asset definitions as described in the requirements in section 3.2. Therefore, these methods in mediation modules can be implemented generically, i.e., be hand-coded. No asset specific code is needed, therefore there is no need for generator.

As appears from the requirements *Mediation Module* should be able to delegate calls to the private and public client modules. Thus, *Mediation Module* should store references to these client modules as attributes. To provide this functionality the component should be set up as depicted in figure C.1:

There is only one method in `Module` interface that require non-generic implementation:

```
AssetClass getClass(String className);
```

This method should return the object of an asset metaclass (more about `AssetClass` in the section 4.4.2) with the given name residing in the corresponding module (in case of *Mediation Modules* the `PersMediationAssetClass`) implementing the `AssetClass` interface. This means that the realization of this method should be generated by the generator depending on the names of asset classes in the asset schema file.

To combine generic and non-generic part inheritance is used as shown on figure C.2.

In the current design I defer the non generic part of *Mediation Module* to a method called `initMetaModel()` which is called by `init()` method. The `init()` method is called when the module is initialized (see appendix C.3).

4.4.2 Asset Class

The object of class Asset Class are used for providing runtime information about asset classes as they are defined in asset schema files, i.e., asset characteristics, relationships, superclasses and modules managing the classes.

As was said in section 4.3 each module has to know what asset classes it is responsible to manage, also each asset should know by which module it is managed. The class diagram in appendix C.3 shows how these responsibilities are fulfilled by the classes in the current design.

The general idea behind this is that `Module` is responsible for knowing and retrieving information about asset metaclasses. It stores a `Map` with pairs "asset name - asset class" for all `AssetClass` objects, describing the classes defined in the asset schema file. This `Map` is initialized at the moment of component initialization when the conceptual content management framework calls the `init()` method of the module. At that moment the non-generic `initMetaModel()` is invoked inside the `init()` method. The `initMetaModel()` method is non-generic and its implementation depends on the current asset definitions. Thus this method initializes the asset metaclasses according to the asset definitions.

If the module needs to get a metaclass with the given name its method `getClass(String name)` is used that retrieves the `AssetClass` object from the `Map` with the given name.

If an asset wants to get information about its metaclass, the its `getType()` method is used, which in its turn calls the `getClass()` method on the module associated with the asset instance.

These interaction scenario is shown in figure 4.5.

4.4.3 Wrappers and Unwrappers

As follows from sections 4.3 and 4.4.2 each module type is coupled with the corresponding `AssetClass` describing the assets residing in the module and powerful enough to be able to instantiate new assets and new query objects. This explains why it is impossible to have only one implementation of `AssetClass` for all modules.

On the other hand the call delegation to the base modules is the part of the functionality of all modules (except for the bottommost ones). The returned, as the result of delegation, objects are of the `AssetClass` of the below module. To return these objects as the response the module must *wrap* the objects. Similarly, when a

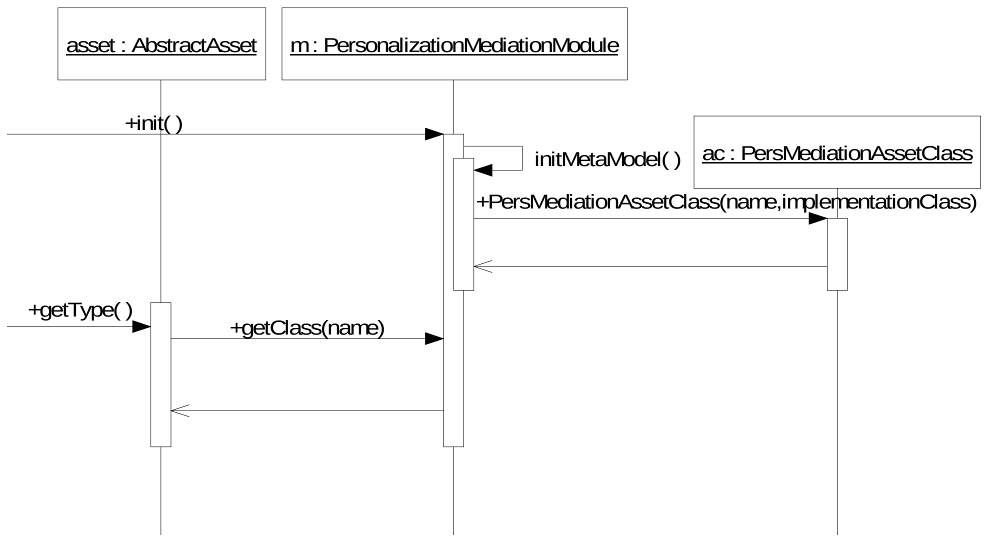

asset : AbstractAsset
m : PersonalizationMediationModule
ac : PersMediationAssetClass
+init( )
initMetaModel( )
+PersMediationAssetClass(name,implementationClass)
+getType( )
+getClass(name)

Figure 4.5: `AssetClass` Retrieval Scenario

module gets an object as a request parameter, before passing it as a parameter of the delegated call the object has to pe *unwraped* to correspond to the `AssetClass` of base module. These operations are very common in many layered architectures (e.g. 8-layered OSI model).

In the current solution these `wrap()` and `unwrap()` operations are implemented as methods of `PersonalizationMediationModule` class, though there can be other possible solutions.

4.4.4 Realization of Asset Role Interfaces

There are five generic and five non-generic interfaces which are used to manipulate single asset instances. These interfaces reflect different states during its life cycle in which asset instance can exist. The description of the interfaces is provided in appendix B.1. These states are called *roles.* The interfaces describing the *roles* of asset instances are called *Role Interfaces* or *Life Cycle Interfaces.*

The figure also shows C.4 the whole *Role Interface* hierarchy. In the figure one can see three different packages: two containing interfaces and one with the implementation of these interfaces.

One of the packages with interface contains only generic interfaces the other contains non-generic. Easy to see from the diagram that interface hierarchies in both packages have identical structure. The hierarchy inside the package corresponds to the conceptual hierarchy of asset roles in the sense that for example any volatile state is a substate of mutable state.

Another type of generalization relationships can be seen between the two packages: generic and non generic. The meaning of these generalizations is quite different. Each class in non-generic package is a subclass of a class in generic package. It is a kind of “package inheritance” or “component inheritance”, which is used to separate all generic asset behavior from the definitions specific.

This is again an example of separation of concerns. There is one more concern, which is not described here: the asset model hierarchies. The issues connected with this inheritance relationship will be discussed further in section **??**.

Because *Role Interfaces* describe the same entity they can implemented by one class as shown in figure C.4. The implementation class stores a reference to an entity in the below module and has to delegate calls to this entity, and wrap and unwrap results and parameters as discussed in section 4.4.3.

Only one method implementation of each was not trivial and needs an explanation is the `accept(LifeCycleVisitor)` method to distinguish between assets in different life cycle states. The usual way to implement accept methods to be used in *visitor pattern* is just calling `visit` method on the parameter passed to the accept method and passing to the parameter the reference to the self class, e.g.

```
class Subclass extends Baseclass{
    public Object accept(SomeBaseClassVisitor visitor) {
        return visitor.visit(this);
    }
}
```

In case of `accept(LifeCycleVisitor)` the situation is different because it is used not to distinguish among subtypes, and thus should not return the instance of the implementation class, but an instance cast to the *Role Interface* corresponding to the life cycle state of the asset instance of the base module. Thus, to implement this functionality first the accept method of the base instance is called to retrieve its life cycle state, and then the instance of the self class cast to the corresponding interface is returned. To call the *accept* method of the base asset a simple *Visitor Class* implementing `LifeCycleVisitor` interface has to be constructed and passed as the parameter of the *accept* method.

4.4.5 Query Interface

As was already said in section 4.3 query objects provide easy way to construct and execute complex queries. The functionality of query objects is similar to the functionality of *Module classes*. Thus, the class diagrams look similar too (see figure C.5). In the `execute()` method of *Query Interface* executes the query and returns the result as an iterator over the result set. The other methods are used for constraints construction.

4.4.6 Iterator Interface

As was already said, the result of execution of the `execute()` method of query objects is the iterator, implementing type-specific variant of *Iterator Interface*. But this is not the only use of iterators. Many methods of *Module Interface* require as parameters or return the results in form of iterators implementing the generic version of *Iterator Interface* `AssetIterator`.

In general the *Iterator Interface* is used to iterate over a set of asset objects. It is usually initialized by a collection of objects and provides a method that returns the next object in this collection until the end of the collection. Thus from the point of view of *Pipes and Filters* architectures (see [SG96]) a collection can be considered as *Data Source*, the output sequence of consequent invocation of `next()` method a *Data Stream* and an iterator as a *Pipe* that delivers the stream of data. The client that invokes the iterator's `next()` method would be considered as *Active Filter* that initiates the whole process and the collection where the data is stored by the client user would play a role of *Data Sink*. To introduce some processing step on data streams additional *filters* are usually introduced between the *DataSource* and *DataSink*, that accept an incoming data stream, process one by one the data objects and produce an output stream. If the role of incoming data stream plays one iterator, then the role of the filter can play the second iterator, which is initialized by the first one. In this case when the `next()` method of the "filter iterator" is called, the "filter iterator" in its turn calls the `next()` method of the "input iterator" and then performs some operation on the object returned by this operation. The result of this operation is returned as the response object of the `next()` method of the "filter iterator". Thus iterator plays a role of *Passive Filter.*

The *Join Filter* can also be represented by an iterator, which is initialized by two other iterators, thus performing the role of two input *Pipes.*

The current design approach makes use of the above described analogies by introducing several generic iterator classes:

Wrapping Iterator (`WrappingIterator`) before returning the next asset object wraps it by calling the method `PersMediationModule.wrap(AbstractAsset)` described in section 4.4.3;

Unwrapping Iterator (`UnwrappingIterator`) before returning the next asset object unwraps it by calling the method `PersMediationModule.unwrap(AbstractAsset aa)` described in section 4.4.3;

Merging Iterator (`MergingIterator`) is initialized by two *AssetIterators* and produces the iterator over the result set according to the merging rules introduced in section 3.2.2.

In the realization of *Pipes and Filters* the decorator pattern is often used, to make use of filter recombination. The same pattern is used here for the design of the above described iterator classes (for the class diagram see appendix, figure C.6).

The described generic iterator classes are widely used for the realization of the module interface in the methods that either take iterators as the parameters or return the result in form of an iterator and have to wrap and unwrap instances for proper call delegation as described in section 4.4.3. The possibility to combine these generic iterator classes is exploited in `lookfore()` operations that must not only wrap and unwrap iterated objects but also merge the results from the *public* and *private* modules.

The *Query Object's* `execute()` method functionality is quite similar to `lookfor()` operation and also needs to merge the results from *public* and *private Query Objects*, therefore the iterators used in the *Query Objects* have to provide the same functionality as the `MergingIterator`, so in the current solution the implementation class of the *Iterator Interface* extends the `MergingIterator` as shown in figure C.6.

4.4.7 Visitor Interface

Visitor Interface is also an interface generated by the *API Generator* but unlike all other non-generic interfaces described above requires no implementation by mediation modules, but is used by client software developers to define it according to their needs and use as visitors in *Visitor Patterns.* The visitors implementing *Visitor Interface* are used as parameters of `accept()` methods defined in *Abstract Asset Interface.* The implementation of these `accept()` methods in implementation class is trivial as defined in *Visitor Pattern*, i.e., invocation of visit method of the visitor object, passed as the parameter.

Chapter 5

Implementation

In the previous chapter the design of the *Mediation Modules* was introduced. Some classes of the designed *Mediation Modules* are generic and were hand-coded. The other parts depend on the asset definitions and are to be generated by the *Mediation Module Generator*. Implementing the generator was broken then into the following steps:

1. The implementation of the generic parts of the *Mediation Modules*. These are the following classes:

   - `PersMediationModule`
   - `PersMediationAssetClass`
   - `MergingIterator`
   - `WrappingIterator`
   - `UnwrappingIterator`

   These classes are implemented inside `de.tuhh.sts.cocoma.personalization.generic` package.

2. The hand-coded implementation of the classes requiring non-generic implementation.

3. Testing the combination of these two parts.

4. Generator development for generating non-generic classes. In parallel testing if the generator output is the same as the hand-coded variant.

The first two steps are not discussed in the current report due to a prescribe design specification given in the previous chapter.

5.1 Several Words on the Generator Design

Although the *Mediation Modules* represent quite a complex piece of software with complex structure of classes, the structure of the piece of software that is responsible for generating this complex class structure, i.e., the *Mediation Module Generator*, does not have this structure. The *Mediation Module Generator* generator consist of only one class, called `PersMediationGenerator` that is responsible for generation of all non-generic parts of the *Mediation Modules*.

The *Mediation Module Generator* (will be referred as *Generator*) is designed to be run inside the *Conceptual Content Management Compiler Framework* to benefit from the *API Generator Symbol Table*(see section 2.2.6) and the *Intermediate Model*. The *API Generator Symbol Table* holds the all information about the interfaces to be implemented. The *Intermediate Model* is created by the *Compiler Parser* and stors the asset definitions (as defined in the schema file) in the internal object model. If the first one, *API Generator Symbol Table*, can be of no use for some generators, i. e. those that do not produce Java code, the second one, *Intermediate Model*, must be used by every generator, that produces any output based on the asset definition schema file.

To be run inside the *Conceptual Content Management Compiler Framework* the *Mediation Module Generator* uses the *Subclassing Interaction Mechanism* common for many *Framework Architectures* (see [SG96]). Therefore, the *Generator* has to subclass the abstract class `de.tuhh.sts.cocoma.compiler.generators.Generator` and to override four abstract *callback methods*[1] defined by this class:

-
  ```
  Collection<ParameterDescription> getRequestedParameters (
          IntermediateModel im)
  ```

  The return values are of type

  ```
      public interface ParameterDescription<T> {
          T getDefaultValue();
          String getDescription();
          String getName();
          Class getType();
  }
  ```

  Answers the parameters needed by the current generator. They have to be provided in the `generate()` call.

-
  ```
  Collection<SymbolTableDescription> getRequestedSymbolTables (
          IntermediateModel im)
  ```

  Returns the types and names of symbol tables needed by this generator. From this dependency information a possible sequence of generator runs is computed. Return values are of type:

  ```
      public abstract class SymbolTableDescription {
          String getName();
          Class getType();
      }
  ```

  stored in the collection.

-
  ```
  SymbolTableDescription getProducedSymbolTable (
          IntermediateModel im)
  ```

  Returns the type and name of the symbol table which will be produced on `generate()`.

-
  ```
  SymbolTable generate (
          IntermediateModel im,
          SymbolTable [] symTabs,
          Map<String,? extends Object> params)
          throws GeneratorException
  ```

[1] In *Framework Architectures* methods provided by the client software to be called by the framework are usually called *Callback Methods*(see [SG96]).

The actual performance method. Takes the intermediate model and the requested symbol tables as parameters. Returns the symbol table created by this generator. As a "side effect" of this method files are created etc.

The parameter of the type `IntermediateModel` passed to all these methods is used if the functionality of the methods depend on asset definitions.

Basically, these methods have to be implemented when developing a generator. Implementation of these methods is necessary and sufficient for generator implementation.

5.2 Java Code Generation Toolkit

The main purposes of the *Mediation Module Generator* are to produce the symbol table and the implementation code of the Java classes defined in chapter 4 in form of text files. Thus, the symbol table of this generator should store the information that can be useful for other generators the purpose of which is to extend or somehow transform the *Mediation Module Generator* output code. Because the output code of this generator basically represents Java classes, the information about these Java classes (names of the classes, implemented interfaces, methods, implementation of methods, etc.) should be stored in the symbol table. To store this information a convenient form has to be found, otherwise defined.

The task of providing the meta-information about classes is not new. It arose with the idea of providing *reflection mechanism.* Java reflection mechanism uses the package `java.lang.reflect` of the standard Java libraries. The classes of this package can be used to retrieve all necessary information about Java classes, but they have no means to perform the creation of Java classes, so cannot be used for the creation of the meta-class information for the symbol table of the *Mediation Module Generator.*

Because no existing library similar to `java.lang.reflect` and providing the possibility to create classes was found, this library had to be created by the developers of the *Conceptual Content Management Compiler Framework.* This library is called Java Code Generation Toolkit and offers classes comparable to those found in the package `java.lang.reflect`, but in contrast to those allowing to manipulate and create classes, as well as to generate the textual representation of these classes in the form Java code. This is implemented by the overriding of `Object.toString()` for these classes.

5.3 Implementation of the Generator

As we defined which methods are to be implemented by the *Generator* and the main sources storing data necessary for the implementation, i.e., *Intermediate Model* and *API Symbol Table*, we can proceed to the discussion of how the set of necessary methods is implemented.

5.3.1 Implementation of `getRequestedParameters()` method

To run the Framework a configuration file has to be provided, which among other configuration details of the Framework gives the possibility to provide input parameters to the generators running in the Framework. The description of these parameters is constructed in the `getRequestedParameters()` method. The implementation of this method consist of creation of objects of type `ParameterDescription` and their initialization according to the names, types and the default values of the parameters. These descriptions and returned from the method in form of a collection.

5.3.2 Implementation of `getRequestedSymbolTables()` method

The implementation consists in construction of a collection of objects of class `SymbolTableDescription`, where each object is initialized by the name and the type of the symbol tables that the generator needs to use during its work. The constructed collection is then returned as the response. The *Mediation Module Generator* depends only on the work of *API Generator*, that generates the interfaces which have to be implemented by any *Module* of the *Conceptual Content Management System*, and, thus, by the *Mediation Module* as well. So the collection returned by the method will contain only one object of class `SymbolTableDescription`.

5.3.3 Implementation of `getProducedSymbolTable()` method

The method has to return the description of the symbol table produced by the *Generator* as an object of class `SymbolTableDescription`. This object has to be initialized by the *name* of this symbol table, chosen by the programmer and the *type*, i.e., the object of class `Class`, which is the metaclass of the implementation class of the symbol table. For the *Mediation Module Generator* the name "`PersMediationGeneratorSymbolTable`" was assigned and the implementation class `PersMediationGeneratorSymbolTable` implemented.

This class `PersMediationGeneratorSymbolTable` is implemented as an inner class of the class `PersMediationGenerator` and subclasses the `SymbolTable` class of the *Framework*. The `PersMediationGeneratorSymbolTable` stores `HashMap`'s with pairs, coupling the asset classes, as defined in the *Intermediate Model*, and the corresponding Java classes as should be generated by the *Generator*, or asset classes and Java methods of some classes, or asset classes and Java fields etc. For example:

```
private HashMap<AssetClass, JavaClass> assetClassToQueryClass =
        new HashMap<AssetClass, JavaClass>();
private HashMap<AssetClass, JavaField> assetClassToModuleField =
        new HashMap<AssetClass, JavaField>();
```

The symbol table also provides the access methods to these `HashMap`'s.

5.3.4 Implementation of `generate()` method

The `generate()` generates the Java source code for the classes defined in the previous chapter. Because the generation is a somewhat long algorithmic process, the structured programming approach for structuring its functionality is used, i.e., a set of methods responsible for the generation of separate classes or class's methods was defined, which are called from the "main" `generate()` method.

According to chapter 4 there are two types of interfaces generated by the *API Generator*: "one-per-model" interfaces and "one-per-asset" interfaces. Generation of the second ones requires the iteration over the set of asset classes. Also, in case of inherited classes it is often desirable to generate the implementation of a base class, save it to the symbol table and then to generate the implementation of the subclass using the information about its parent saved in the symbol table. This requires the iteration over a hierarchy of classes. For the hierarchical iteration the `de.tuhh.sts.cocoma.compiler...hierarchy.HierarchyNode` class is used.

To facilitate the generation the *Java Code Generation Toolkit*(see section 5.2) is used.

5.4 Generator Configuration

As was already said the *Conceptual Content Management Compiler Framework* requires a configuration file with the configuration information about the generators to run. The configuration file for the generation of *Mediation Modules* is shown in appendix D. This file defines the class names of the *scanner* and *parser* used in the *compiler frontend* and the *configuration* of the generators used in the *backend*. The *configuration* consist of tow generators "persmedgen"(the *Mediation Module Generator*) and "apigen"(the *API Generator*). The the names of the classes implementing the generators are correspondingly `de.tuhh.sts.cocoma.compiler.generators.persmedgen.PersMediationGenerator` and `de.tuhh.sts.cocoma.compiler.generators.api.APIGenerator`. For each generator a parameter list is provided. For the *Mediation Module Generator* these parameters are the directory where the generated code is saved and the package in which the generated classes will reside.

Chapter 6

Summary and Future Work

6.1 Summary

The overview of the literature made within the scope of the current work proved the personalization to be an important issue for different types of software systems including Conceptual Content Management Systems. One of the problems of implementation of Personalization is the need to link the public and the private view to the system. The general approach to solve this problem is to provide some piece of software, responsible for linking these two views, i.e., mediating between them. During the current work the attempt to solve the problem of mediation between public and private modules of Conceptual Content Management Systems was made. The analysis of the problem showed that it is not possible to do this generically. Therefore, the Generator for the generation of Mediation Modules was designed and implemented.

The Generator and a set of generic classes solve the mediation task for the Conceptual Content Management Systems. They are realize the full functionality needed to facilitate personalization of public assets, though some bugs can still be found.

The Mediation Modules generated by the Generator developed were tested and proved to work on a simple asset definition model, containing two asset classes, one of which inherits from the other. The developed software still needs to be tested on more complex models.

The value of this project consist in the first implementation of the powerful Personalization mechanism ensuring openness and dynamics of Conceptual Content Management Systems.

6.2 Future Work

The current project also showed the directions to the future work on the way to building personalized open and dynamic applications using Conceptual Content Management systems.

6.2.1 Hiding the Origin of Personalized Asset

Modification of Asset Schema Definitions by introducing the relationship to origin, brings to the generation of access methods for this relationship. Therefore, users get the possibility to see and modify this relationship. But it should not see this relationship at all. The Problem can be solved by introducing an additional Transformation module, which transforms the component behavior according to the schema with origin, to the behavior without visible origin field, i.e. transforms schema including origin relationship to a schema without it.

6.2.2 Asset Publishing

The current solution facilitates only the content personalization, i.e., transferring the public assets to private view or creation of assets in private view. The content publishing as the transformation of private assets to public view is the revers operation, so can be using the similar mediation modules. When the new content is published the previously public version of the content should be transformed to the views of all users. The users in this case may accept or decline changes. In case of accepting some operation merging the two versions have to be introduced.

6.2.3 Structure Personalization

The current solution facilitates only the content personalization. To provide the possibility for the personalization of the asset structure a problem of matching public and private schema has to be solved. This can be done by implementation of transformation modules, matching public and personalized schema.

Appendix A

Component Configuration

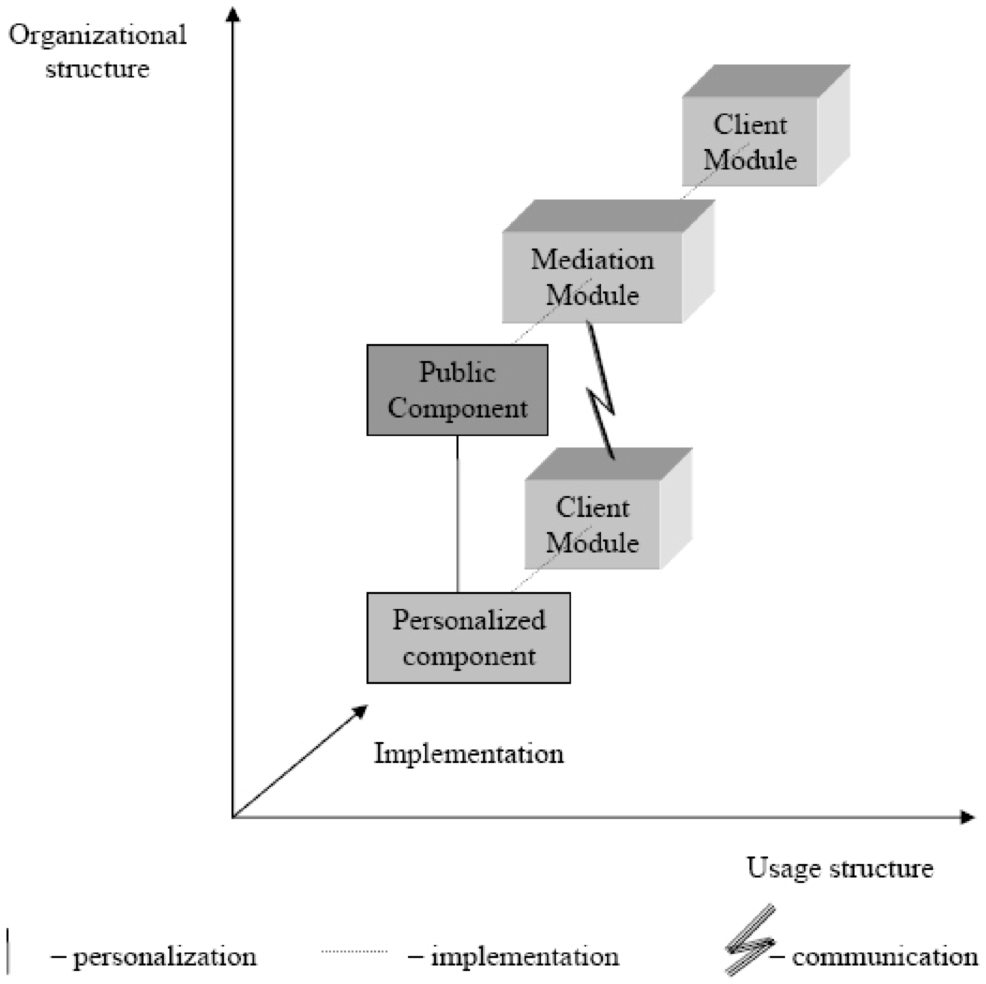


Figure A.1: Component Configuration for Personalization

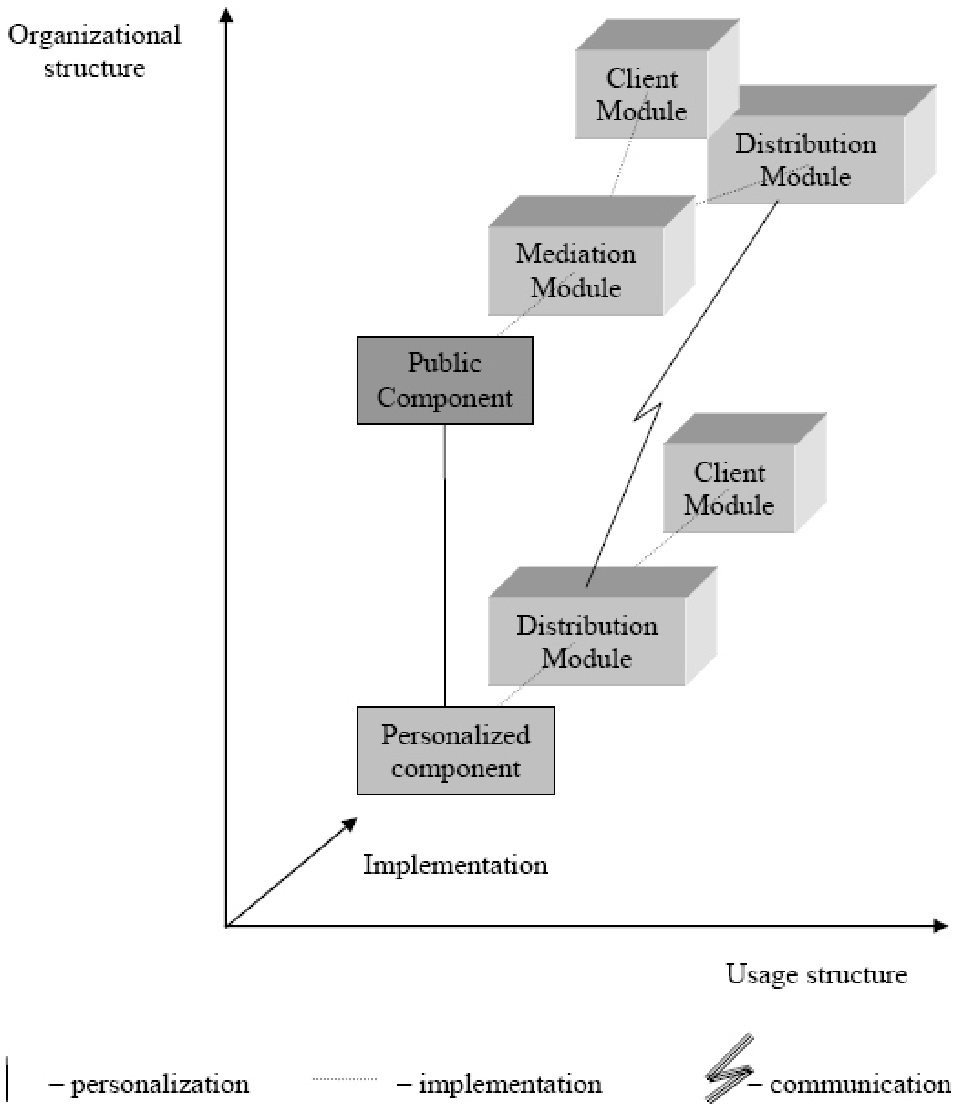


Figure A.2: Module Configuration for Personalization

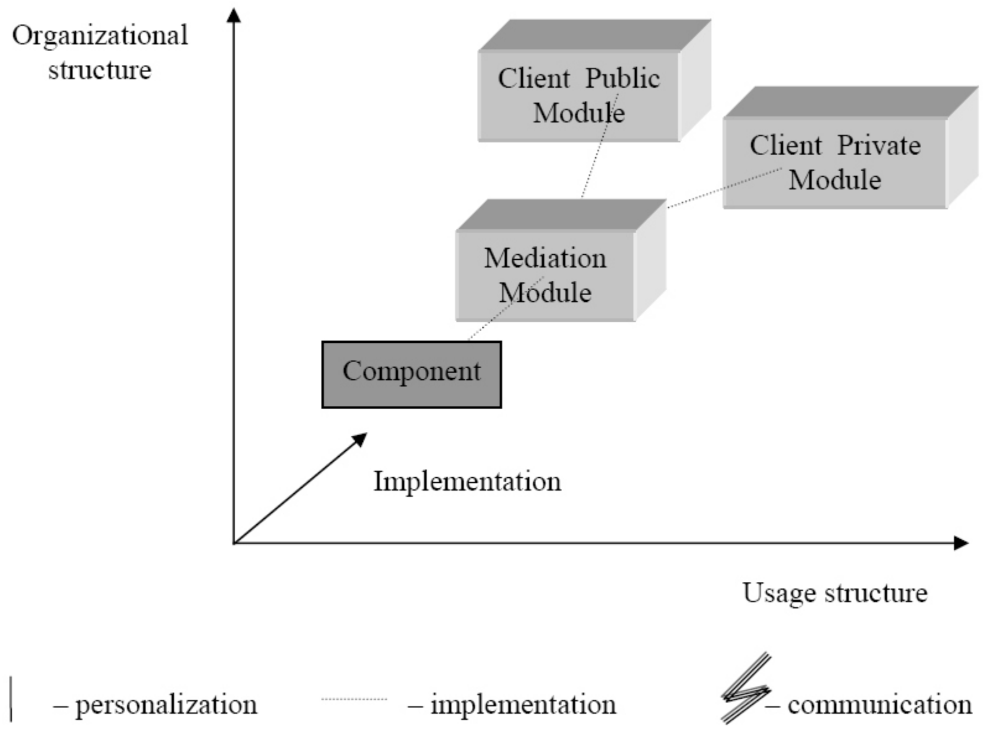


Figure A.3: Module Configuration for Personalization

Appendix B

Interface Specification

Table B.1: Interface Specification

Interface	Interface Type	Interface Name	Implementation Class	Main Methods	Interface Description
Abstract Interface (General Base Interface)	generic	AbstractAsset	Person	ID getID() AssetClass getType() Object accept(LifeCycleVisitor) void addLifeCycle-Listener(LifeCycleListener) void removeLifeCycle-Listener(LifeCycleListener)	includes all methods not depending on the asset definitions in schema file and which have meaning and can be invoked independently of the asset life cycle; these are the methods for asset retrieval by ID, method for getting asset metaclass, *accept()* method used in the visitor pattern to retrieve the asset life cycle state, methods for addition and deletion of life cycle listeners activated when asset state changes
	non-generic	AbstractPerson		String getName() AbstractPerson getOrigin() Object accept(PersonVisitor)	includes all methods, which have sense and can be invoked independently of the asset life cycle, but depend on asset definitions in schema file, these are the *get..()* methods for all asset attributes (the names of the methods corresponds to the names of asset attributes) and *accept()* method used in the visitor pattern to distinguish among asset subclasses
Persistent Interface	generic	Asset		MutableAsset lockAsAsset()	reflects the methods specific to the asset in persistent state
	non-generic	Person		MutablePerson lockAsPerson() or MutablePerson lock()	reflects the type-safe variant of the methods special for the asset in persistent state

Abstract Mutable Interface		Abstract-MutablePerson		setName(String)
				setOrigin1(AbstractPerson)
					base interface for assets in all mutable states; contains methods for modification of asset attributes

Mutable Interface	generic	MutableAsset		Asset abortAsAsset() Asset commitAsAsset() NewAsset deleteAsAsset()	corresponds to the locked state of an asset; provides methods for transferring asset to persistent state aborting or committing introduced changes, and method for the deletion of persistent asset instance
	non-generic	MutablePerson		Person commitAsPerson() Person abortAsPerson() NewPerson deleteAsPerson()	corresponds to the locked state of an asset and provides type-safe variants of the methods transferring asset either to persistent or to volatile state.
New Interface (Volatile Interface)	generic	NewAsset		Asset storeAsAsset()	corresponds to the asset in volatile state
	non-generic	NewPerson		Person storeAsPerson()	corresponds to the asset in volatile state, provides type-safe variant of *NewAsset* interface

Continued on the Next Page. . .

Interface	Interface Type	Interface Name	Implementation Class	Main Methods	Interface Description
Query Interface	generic	AssetQuery	PersonQuery	constrainBySubQuery(AssetQuery) constrainByQueryConstraint(-Module.QueryConstraint) AssetIterator executeForAsset()	base generic interface for querying objects; can be used to construct queries containing sub-queries, but usually its subclasses are more convenient
	non-generic	PersonQuery		constrainName...(String) constrainOrigin1...(AbstractPerson) PersonIterator executeForPerson() or PersonIterator execute()	inherits from *AssetQuery* interface; a powerful interface to construct and execute complex queries; provides methods for constraining all asset attributes and type-safe method for query execution
Iterator Interface	generic	AssetIterator	PersonIterator	int getLength() AbstractAsset nextAsset()	interface for collections of asset objects; extends standard Java *Iterator* interface and defines methods for retrieving the number of asset objects in collection and *nextAsset()* method returning next instance of *AbstractAsset* type
	non-generic	PersonIterator		AbstractPerson nextPerson()	inherits from *AssetIterator*; defines a type-safe variant of method for iteration over the asset collections
Factory Interface		Personalization-AssetFactory	Personalization-AssetFactory	NewPerson createPerson()	interface used to create asset objects of the least specific type in the asset hierarchy
		PersonFactory	PersonFactory	NewStudent createStudent()	interface used to create asset objects of the more specific type in the asset hierarchy
Visitor Interface		Personalization-TypeVisitor or PersonVisitor	no implementation required	Object visit(AbstractPerson)	used in *Visitor Patterns*; need no implementation; must be defined by system users according to their needs; used to distinguish among subclasses of the base class
Module Interface	generic	Module	Personalization-MediationModule	create(...) modify(...) delete(...) lookfor(...) getClass(...)	one per asset model, i.e., one per component; defines set of methods to create, modify and retrieve assets as well as a method for getting the asset metaclass object with by the name of the asset class as defined in the asset schema
Asset Metaclass Interface)		AssetClass	PersMediationAsset-Class	String getName() AssetIterator definedAttributes() AssetClass getSuperClass() NewAsset createInstance() AssetQuery startQuery()	asset metaclass interface, defining methods for getting information about asset class name, its superclass and defined attributes as well as a method for creating new instance of the class with default attributes and a method for initialization of query object for the asset class

Appendix C

Interface Realization: Class Diagrams

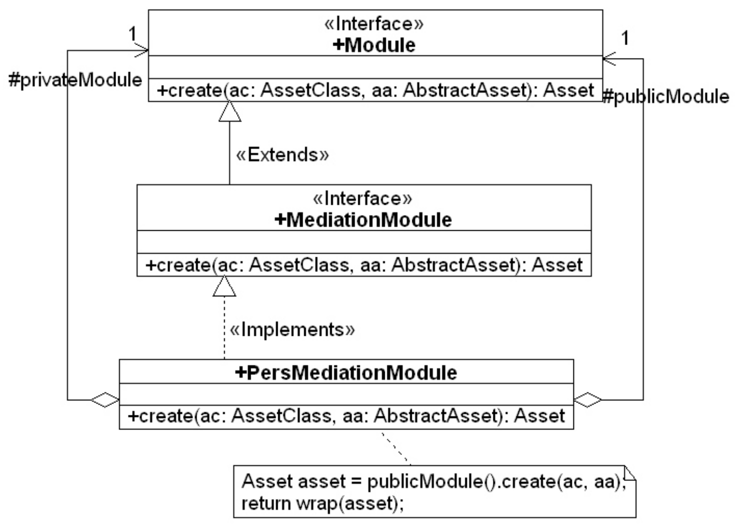
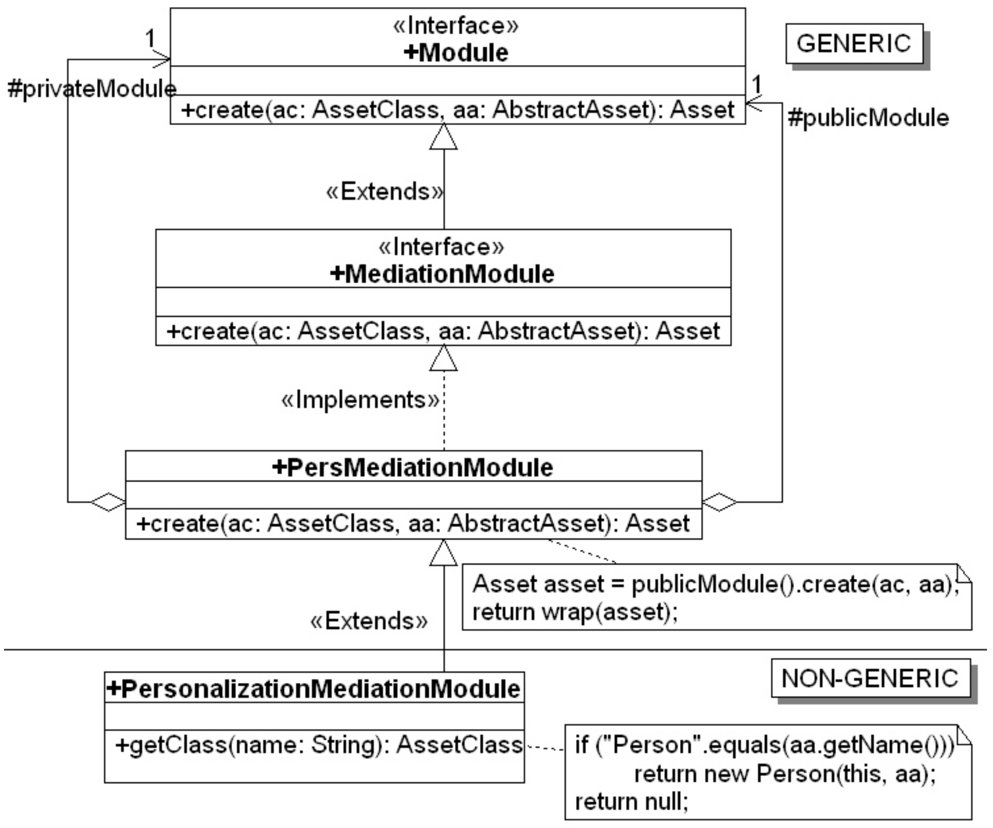

«Interface»
+Module
+create(ac: AssetClass, aa: AbstractAsset): Asset
1
#privateModule
1
#publicModule
«Extends»
«Interface»
+MediationModule
+create(ac: AssetClass, aa: AbstractAsset): Asset
«Implements»
+PersMediationModule
+create(ac: AssetClass, aa: AbstractAsset): Asset
Asset asset = publicModule().create(ac, aa);
return wrap(asset);

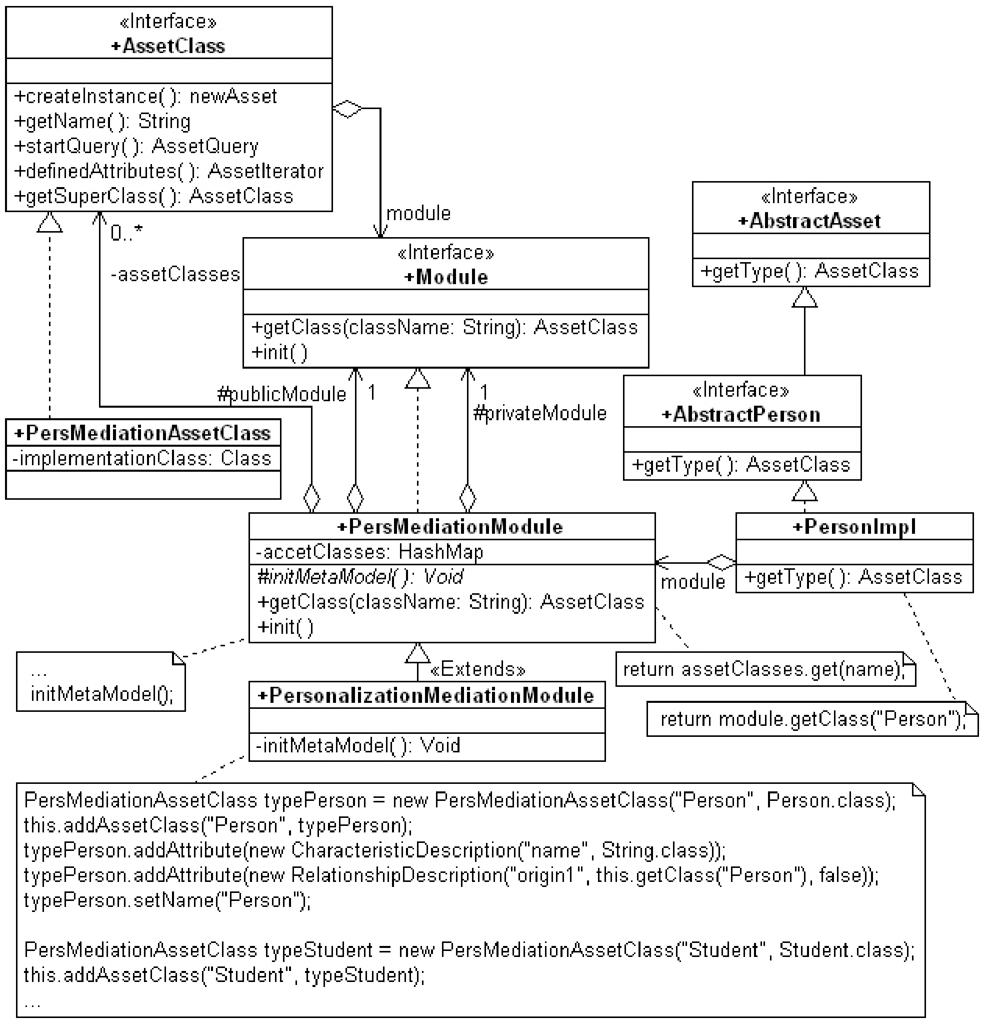
Figure C.1: Implementation of `Module` Interface: Generic Part


«Interface»
+Module
+create(ac: AssetClass, aa: AbstractAsset): Asset
GENERIC
1
#privateModule
1
#publicModule
«Extends»
«Interface»
+MediationModule
+create(ac: AssetClass, aa: AbstractAsset): Asset
«Implements»
+PersMediationModule
+create(ac: AssetClass, aa: AbstractAsset): Asset
«Extends»
+PersonalizationMediationModule
+getClass(name: String): AssetClass
NON-GENERIC

```
Asset asset = publicModule().create(ac, aa);
return wrap(asset);
```

```
if ("Person".equals(aa.getName()))
        return new Person(this, aa);
return null;
```

Figure C.2: Implementation of `Module` Interface: Generic and Non-Generic Parts



```
...
initMetaModel();
```

```
return assetClasses.get(name);
```

```
return module.getClass("Person");
```

```
PersMediationAssetClass typePerson = new PersMediationAssetClass("Person", Person.class);
this.addAssetClass("Person", typePerson);
typePerson.addAttribute(new CharacteristicDescription("name", String.class));
typePerson.addAttribute(new RelationshipDescription("origin1", this.getClass("Person"), false));
typePerson.setName("Person");

PersMediationAssetClass typeStudent = new PersMediationAssetClass("Student", Student.class);
this.addAssetClass("Student", typeStudent);
...
```

Figure C.3: Asset Class Implementation: Initialization and Retrieval of Metainformation about Asset Classes

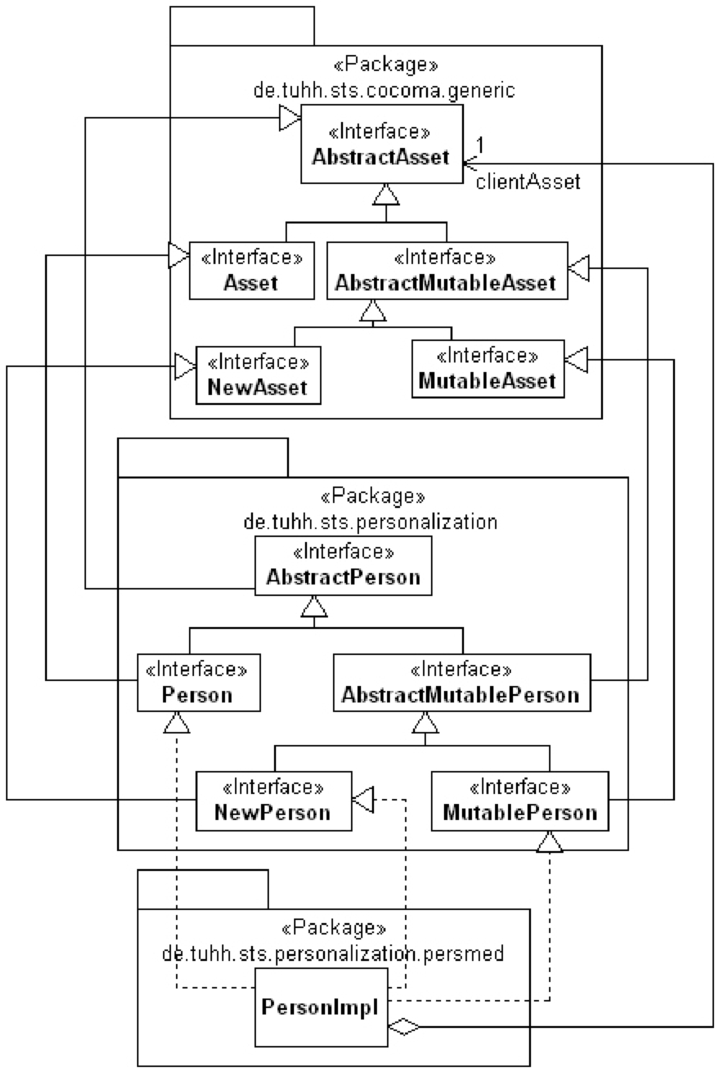
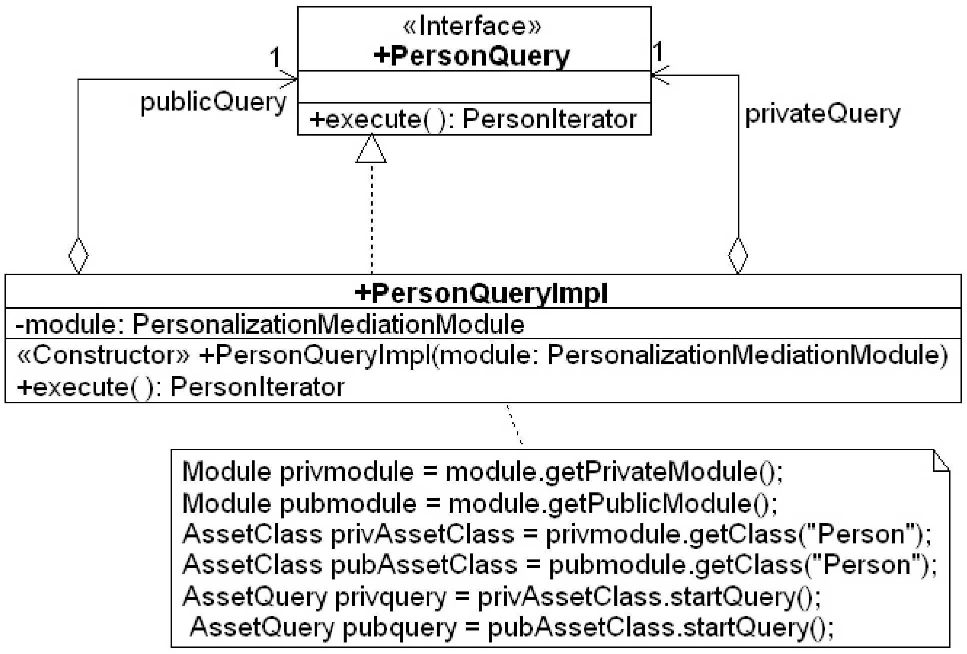
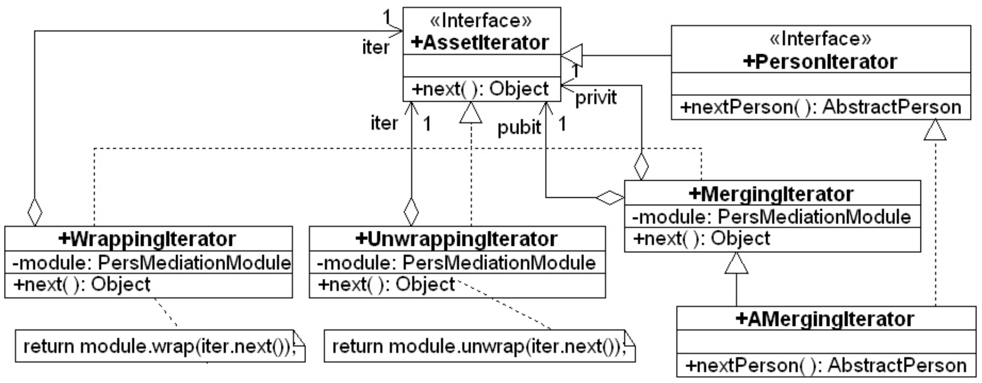


Figure C.4: Asset Roles


«Interface»
+PersonQuery
+execute( ): PersonIterator
1
publicQuery
1
privateQuery
+PersonQueryImpl
-module: PersonalizationMediationModule
«Constructor» +PersonQueryImpl(module: PersonalizationMediationModule)
+execute( ): PersonIterator

```
Module privmodule = module.getPrivateModule();
Module pubmodule = module.getPublicModule();
AssetClass privAssetClass = privmodule.getClass("Person");
AssetClass pubAssetClass = pubmodule.getClass("Person");
AssetQuery privquery = privAssetClass.startQuery();
 AssetQuery pubquery = pubAssetClass.startQuery();
```

Figure C.5: Implementation of Query Interface



```
return module.wrap(iter.next());
```

```
return module.unwrap(iter.next());
```

Figure C.6: Implementation of Iterators

Appendix D

Mediation Module Generator Configuration File

```
<?xml version="1.0"?>
<catxmlns:util="http://www.sts.tu-harburg.de/2004/java/util/xmlconfigfile">
   <scanner class="de.tuhh.sts.cocoma.compiler.ADLScanner"/>
   <parser class="de.tuhh.sts.cocoma.compiler.ADLParser"/>

   <configuration name="persmedgen">
       <param name="outputDirBase">gen</param>
       <generator name="persmedgen" class=
               "de.tuhh.sts.cocoma.compiler.generators.persmedgen.PersMediationGenerator">
           <param name="outputDir">
               <util:xpath path="../../../param[@name='outputDirBase']/text()"/>/src
           </param>
           <param name="targetPackage">
               de.tuhh.sts.personalization.persmed
           </param>
       </generator>
       <generator name="apigen" class=
               "de.tuhh.sts.cocoma.compiler.generators.api.APIGenerator">
            <param name="outputDir">
               <util:xpath path="../../../param[@name='outputDirBase']/text()"/>/src
            </param>
            <param name="targetPackage">de.tuhh.sts</param>
       </generator>
   </configuration>
</cat>
```

Bibliography

[Bat97] Don Batory. Intelligent components and software generators. Technical Report CS-TR-97-06, The University of Texas at Austin, Department of Computer Sciences, April 1 1997. Mon, 28 Apr 103 21:07:00 GMT.

[Bat03] Don S. Batory. The road to utopia: A future for generative programming. In Christian Lengauer, Don S. Batory, Charles Consel, and Martin Odersky, editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2003.

[Bat04] Don S. Batory. Program comprehension in generative programming: A history of grand challenges. In *International Workshop of Program Comprehension*, pages 2–13. IEEE Computer Society, 2004.

[BB03] Alex Borgida and Ronald J. Brachman. Conceptual modeling with description logics. In Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter Patel-Schneider, editors, *The Description Logic Handbook: Theory, Implementation and Applications*, pages 349–372. Cambridge University Press, Cambridge, England, 2003.

[BDG$^+$95] Don Batory, Sankar Dasari, Bert Geraci, Vivek Singhal, Marty Sirkin, and Jeff Thomas. Achieving reuse with software system generators. *IEEE Software*, pages 89–94, September 1995.

[BL01] Paul Browning and Mike Lowndes. Techwatch report: Content management systems. Technical report, The Joint Information System Committee, September 11 2001.

[Blo83] Toby Bloom. *Dynamic module replacement in a distributed programming system.* PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1983.

[BLS98] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: Tools for implementing domain-specific languages. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, page 143, Washington, DC, USA, 1998. IEEE Computer Society.

[BMW82] Alexander Borgida, John Mylopoulos, and Harry K. T. Wong. Generalization/specialization as a basis for software specification. In *On Conceptual Modelling (Intervale)*, pages 87–117, 1982.

[BO92] Don S. Batory and Sean W. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, 1992.

[Boi04] Bob Boiko. *Content Management Bible.* Wiley Publishing Inc., 2nd edition, November 2004.

[Bol03] Susanne Boll. Mm4u - a framework for creating personalized multimedia content. In *Proceedings of the International Conference on Distributed Multimedia Systems (DMS' 2003)*, September 2003.

[Bon03] Stéphane Bonnet. Model driven software personalization. In *Smart Objects Conference (SOC 2003)*, pages 114–117, Grenoble, France, May 15-17, 2003.

[Bro84] M. L. Brodie. On the development of data models. In M. L. Brodie, J. Mylopoulos, and J. W. Schmidt, editors, *On Conceptual Modelling, Perspectives from Artificial Intelligence, Databases and Programming Languages*, pages 19–48. Springer-Verlag, New York, New York, 1984.

[CEG$^+$98] Krzysztof Czarnecki, Ulrich W. Eisenecker, Robert Glück, David Vandevoorde, and Todd L. Veldhuizen. Generative programming and active libraries. In Mehdi Jazayeri, Rüdiger Loos, and David R. Musser, editors, *Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 25–39. Springer, 1998.

[dPLSdF94] J. do Prado Leite, M. Sant'Anna, and F. de Freitas. Draco-PUC: A technology assembly for domain oriented software development. In *Proceedings of the Third International Conference on Software Reuse*, pages 94–100, 1994.

[EV03] Magdalini Eirinaki and Michalis Vazirgiannis. Web mining for web personalization. *ACM Transactions on Internet Technology (TOIT)*, 3(1):1–27, February 2003.

[Gil00] Frank Gilbane. What is content management? *The Gilbane Report*, 8(8), october 2000. `http://gilbane.com/gilbane_report.pl/6/What_is_Content_Management.html`.

[Gua98] N. Guarino. Formal ontology and information systems. In N. Guarino, editor, *Formal Ontology in Information Systems*, pages 3–18. IOS Press, Amsterdam, 1998.

[JPB97] Guillermo Jiménez-Pérez and Don Batory. Memory simulators and software generators. In Medhi Harandi, editor, *Proceedings of the 1997 Symposium on Software Reusability (SSR '97)*, pages 136–145, 1997.

[KA03] Michael Kohlhase and Romeo Anghelache. Towards collaborative content management and version control for structured mathematical knowledge. In Andrea Asperti, Bruno Buchberger, and James H. Davenport, editors, *Mathematical Knowledge Management, Second International Conference, MKM 2003, Bertinoro, Italy, February 16-18, 2003, Proceedings*, volume 2594 of *Lecture Notes in Computer Science*, pages 147–161. Springer, 2003.

[KBA00] Katarzyna Keahey, Peter Beckman, and James Ahrens. Ligature: Component architecture for high performance applications. *The International Journal of High Performance Computing Applications*, 14(4):347–356, Winter 2000.

[Kha00] Latifur R. Khan. *Ontology-based Information Selection*. PhD thesis, Faculty of the Graduate School, University of Southern California, October 17 2000.

[KR02] Nora Koch and Gustavo Rossi. Patterns for adaptive web applications. In *7th European Conference on Pattern Languages of Programs*, October 29 2002.

[Mar02a] Esko Marjomaa. "peircean" reorganization in conceptual modeling terminology. *Journal of Conceptual Modeling*, (23), January 2002. http://www.inconcept.com/jcm/January2002/esko.html.

[Mar02b] Raphael Marvie. *Separation of Concerns and Metamodeling applied to Software Architecture Handling*. PhD thesis, LIFL, Universit des Sciences et Technologies de Lille, February 20 2002.

[McG04] Gerry McGovern. Don't make these mistakes when buying content management software. *New Thinking*, March 2004. `http://www.gerrymcgovern.com/nt/2004/nt_2004_03_29_CMS.htm`.

[Nei80] James Milne Neighbors. *Software construction using components*. PhD thesis, University of California, Irvine, October 09 1980.

[Nei89] James M. Neighbors. Draco: A method for engineering reusable software systems. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability – Concepts and Models*, volume I, chapter 12, pages 295–319. ACM Press, 1989.

[Nei92] J. Neighbors. The evolution from software components to domain analysis. *International Journal of Software Engineering and Knowledge Engineering*, 2(3):325–354, September 1992.

[Nei01] James M. Neighbors. *Draco 1.2 Users Manual*. Department of Information and Computer Science; University of California, Irvine , CA 92717, October 09 2001.

[Nii04] Markopekka Niinimäki. Conceptual modelling languages. Technical Report A-2004-1, Department of Computer Sciences, University of Tampere, 2004. Ph.D. Thesis.

[PDH99] Allen Parrish, Brandon Dixon, and David Hale. Component based software engineering: A broad based model is needed. In *International Workshop on Component-Based Software Engineering proceedings*, pages 43–46, May 10 1999.

[RSG01] Gustavo Rossi, Daniel Schwabe, and Robson Guimarães. Designing personalized web applications. In *WWW*, pages 275–284, 2001.

[SB00] Yannis Smaragdakis and Don Batory. Application generators. In J. Webster, editor, *Software Engineering volume of the Encyclopedia of Electrical and Electronics Engineering*. John Wiley and Sons, March 29 2000.

[SBK03] Cyrus Shahabi and Farnoush Banaei-Kashani. Efficient and anonymous web-usage mining for web personalization. *Institute for Operations Research and the Management Sciences Journal on Computing*, 15(2):123–147, 2003.

[SBS05] H.W. Sehring, S. Bossung, and J.W. Schmidt. Active learning by personalization - lessons learnt from research in conceptual content management. In Bruno Encarnaca Jose Cordeiro, Vitor Pedrosa and Joaquim Filipe, editors, *Proceedings of the 1st International Conference on Web Information Systems and Technologies*, pages 496–503. INSTICC Press Miami, May 2005.

[Seh03] Hans-Werner Sehring. *Konzeptorientierte Inhaltsverwaltung Modell, Systemarchitektur und Prototypen.* Dissertation, Hamburg University of Technology, November 2003.

[SG96] Mary Shaw and David Garlan. *Software architecture: perspectives on an emerging discipline.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[SS03] Joachim W. Schmidt and Hans-Werner Sehring. Conceptual content modeling and management. In Manfred Broy and Alexandre V. Zamulin, editors, *Ershov Memorial Conference*, volume 2890 of *Lecture Notes in Computer Science*, pages 469–493. Springer, 2003.

[SS04] Hans-Werner Sehring and Joachim W. Schmidt. Beyond databases: An asset language for conceptual content management. In *Proceedings of Advances in Databases and Information Systems*, pages 99–112, 2004.

[TO01] Peri L. Tarr and Harold Ossher. Hyper/J^{TM}: Multi-dimensional separation of concerns for java^{TM}. In *Proceedings of International Conference on Software Engineering*, pages 729–730. IEEE Computer Society, 2001.

[TOHS99] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of International Conference on Software Engineering '99*, pages 107–119, Los Angeles CA, USA, 1999.

[Tri05] Bill Trippe. Component content management in practice. *The Gilbane Report*, February 2005. `http://gilbane.com/whitepapers.pl?view=14`.

[Wid98] Tanya Widen. Formal language design in the context of domain engineering. Master's thesis, (M.S.)–Oregon Graduate Institute of Science and Technology, 1998.

[Wie92] Gio Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, 1992.