Adaptable and Adaptive Visualizations in Concept-oriented Content Management Systems

Hans-Werner Sehring
Experience Design and Emerging Technologies
T-Systems Multimedia Solutions GmbH
Dresden, Germany
hans-werner.sehring@t-systems.com

Abstract—**One task of content management is the publication of content. The necessary means to render content into documents are usually developed alongside other aspects of content management systems, in particular the content's schema. There are content management applications, however, that require open and dynamic content modeling and management. These concept-oriented content management (CCM) systems have been studied carefully. As a consequence, content visualization in this kind of applications has to be adaptive and cannot be statically tailored to one given content structure alone. This paper gives a roundup of CCM, discusses means to abstractly define content visualizations, and presents an approach to adaptive visualization. The paper is an extended version of [1].**

Keywords-concept-oriented content management; adaptive user interfaces; personalization; content distribution

I. Introduction

In practice, there is no sharp definition of content management. There is agreement, though, that content management has to support the separation of layout, structure, and content [2]. To this end a typical *content management system* (*CMS*) allows to define structure through a *content schema* or *content model*, to manage content as data, and to render content into documents following a specified layout through *templates* (static view components plus code for the representation of content) during *playout*.

CMSs are applied in different scenarios, though. In particular there are cases where content is itself the primary entity of interest – when digital content is considered like in digital image collections, video portals, and Web 2.0 applications – and there are cases when content is used to represent real-world entities that cannot adequately be represented by structured data.

For the latter class of applications we have introduced *Concept-oriented Content Management* (*CCM*). In addition to the above-mentioned general requirements, *Concept-oriented CMSs* (*CCMSs*) have to support personalization of both data [3] and schemata [4] as a means to express subjective interpretations of content. Additional requirements follow from these properties: content models have to be open to changes and CCMSs have to follow model changes dynamically, while users need to be able to communicate with each other in the presence of personalized content (models) that may differ to a certain degree.

Earlier papers reported on the technical foundations of CCM that allow handling schema evolution and individualized communication in CCMSs. In this paper we discuss visualization matters for such systems, both for editing content according to personalized models as well as the rendering of content into documents that can be published independently of a content schema.

The remainder of this paper is organized as follows: in Section II we revisit CCM as a content management approach. Since the technical details of the CCM approach have been described thoroughly in other publications [4], the paper gives a summary of these topics. In Section III, however, we provide a more detailed look on content modeling with CCM. Sections IV and V cover the main topic of this paper, adaptive visualization of content. In Section IV adaptable visualizations for CCMSs are discussed in general, and specifically details on view models. Section V discusses the modeling of controllers to handle interaction. The paper concludes in Section VI with an outlook on future research.

II. Concept-oriented Content Management

The CCM approach has been designed for content management applications that require handling content as personalized variants rather than in one standardized form. Two major requirements to CMSs have been identified for this kind of applications: content models have to be open to schema changes (*openness*), and CCMSs have to follow model changes dynamically (*dynamics*).

As a means to meet these requirements, three major contributions have been identified for the CCM approach: a language for open content modeling, a model compiler that translates content schema definitions into CCMSs that both implement a given schema and allow communication between subsystems with different variants of a schema, and a CCMS architecture that allows systems evolution through incremental compilation.

In this section we describe the definition of CCM models and the technology to implement CCMSs.

A. Foundations of Concept-oriented Content Management

Various projects have shown the need for a form of content management that is concerned about content that represents real world entities. This form of content management usually is employed for entities that cannot accurately be described by structured data, e.g., by records in databases. One example for a class of such entities is that of pieces of art. A piece of art can be enjoyed as content, but it also represents the process of its creation, in particular the epoch in that it has been created, the artist and her or his way of living, as well as the message that shall be conveyed by the artwork (an opinion of the artist, or a statement of the employer as it is the case for politically motivated art, advertisements, etc.).

Content that represents entities is – in contrast to structured data – subject to individual interpretations. Content can be stored, shared, etc., but it will necessarily lead to subjective views on the represented entities.

Collaborative work with content that represents entities requires support to make those aspects of interpretations explicit that are intended by the author. In accordance with observations already made by others (in fact, they have been made as early as by Cassirer's works [5]), we propose to provide a conceptual model that accompanies the content.

We call the union of content and a conceptual model that both describe the same real-world entity an *asset*.

In order to be able to express subjective views, CCMSs support personalization, in particular personalization of asset instances, models, and presentations. As already mentioned, they do so by providing model openness and systems dynamics.

B. *Asset Definition Language*

The *asset definition language* (*ADL*) allows expressing entity models as laid out in the previous subsection. In this subsection we give a small glimpse of the ADL syntax, while we discuss modeling with assets in Section III.

Asset schemata are given as *models*. Models contain *classes* with *members* (*content handles* and *attributes*) and instance definitions. Furthermore, classes can be imported from other models, thus allowing model reuse (see Section III).

The following code shows an example of an asset model:

```
model MyModel
from SomeOtherModel import SomeClass
class MyClass
class MySubClass refines SomeClass
let myAsset :MyClass := …
```

Here, two classes and one instance are defined as part of the model called `MyModel`, where one class is defined as a refinement of an existing class imported from another model.

In the `let` statement for the definition of the named instance `myAsset` a type constraint can be seen after the colon. By using such type information a more general type than that of the actual instance can be given.

A type is given by the name of a class. If an asterisk follows the class name then the type refers to a set-valued type over the given base type.

Classes separate the two aspects of an asset – content and concept view – in respectively named compartments:

```
class MyClass refines SomeClass {
  content someContentHandle :HandleType
  concept … ; see below
}
```

Each content handle is given by a name and a type constraint (introduced by the colon). Please also note the semicolon introducing a one-line comment.

Since assets are similar to signs considered in Semiotics, we loosely base the concept part of assets on Peirce [6], in particular his distinction of three description categories.

The first category of the conceptual model consists of attributes that contain values that are inherent to instances:

```
class MyClass {
  concept characteristic c :T
          characteristic d :T2 := … }
```

Characteristic values are not first class citizens of an asset model. The usable types (in the example: `T` and `T2`) are borrowed from an underlying implementation language. Currently, we use Java for this purpose: any Java class from the standard or other class libraries can be used as a type, and Java expressions can be used in initializations (":=").

If an asset can be related to other assets, named and typed relationships can be defined as the second kind of attribute:

```
class MyClass {
  concept relationship r1 :C
          relationship r2 :D* }
```

Here, a relationship `r1` to an instance of asset class `C` and a many-to-many relationship `r2` to instances of type `D` are defined.

The third contribution of class definitions is that of regular definitions on the type level. These apply to all instances of the respective class (and, by means of inheritance, that of subclasses). Of course, classes itself as well as the type constraints are already regular contributions. Nevertheless, the need for application-specific constraints often arises:

```
class MyClass {
  concept constraint constraint1 c = x
          constraint constraint2 c < y
            onviolation … }
```

These definitions define the value of `c` to always be equal to `x` and less than `y` (where the comparison operators are defined in a type-specific way). Changes to the asset that would violate the first constraint are forbidden and lead to runtime errors. The second constraint contains a productive rule that establishes (or at least tries to establish) a situation that conforms to the constraint.

There are seven built-in operators to check for equality ("="), inequality ("#"), ordering ("<", "<=", ">", ">="), and similarity ("~"). These are implemented in a type-specific way. E.g., "<" tests for a subtype for classes, and for a subset for asset sets.

Named asset instances can be referred to by their name. Members of instances can be accessed by the projection operator ("`.`"), e.g., `myAsset.x`. Asset sets are given by a comma-separated asset enumeration in brackets.

The asset creation and manipulation sublanguage controls the lifecycle of asset instances. It allows creating and modifying asset instances through operations like:

```
create MyClass { c := x }
create MyClass someMyClassInstance
modify someAsset { c := x }
modify someAsset someOtherAsset
delete someAsset
```

The statements are available in intentional form (giving member values) and in extensional form (giving a prototype). A set of prototypes can be given in the extensional forms; then statements are applied element-wise.

The asset query sublanguage allows finding asset instances using statements like

```
lookfor MyClass { c # y }
```

Operations can be combined by means of concatenation. The following sample statement updates all instances of `MyClass` with a value `x` of attribute `c` so that `c` becomes `y`:

```
modify lookfor MyClass { c=x } { c:=y }
```

Concatenation follows the implicit rules that (1) sets of sets of instances are flattened to sets of instances, that (2) there is no distinction between singleton sets and single instances, and that (3) projection can be applied to sets element-wise. For example, the statement

```
{lookfor MyClass { c = x },
 lookfor MyClass { c = y }}.c
```

retrieves the union of all instances of `MyClass` with a `c` value of `x` or `y` and projects it to the value(s) of the attribute `c` (this can result to `{}`, `x`, `y`, or `{x,y}`, depending on the existing asset instances).

C. *On Open Content Modeling*

Asset models are units of model reuse. Through the `import` statement classes can be imported and used for two reasons: for domain interrelation and for personalization.

The first way of reuse, domain interrelation, allows integrating definitions in order to use a domain as a related domain or subdomain. If studying art history, for example, one will want to reuse some work of historians.

One specific property of the ADL is its ability to redefine content classes in a specific context. By this means the ADL can be used for personalization and for the management of content revisions and content variants. Imports of classes for this reason are the second use of model reuse.

The redefinition of classes can include the addition of attributes, the removal of attributes, and changes to the inheritance hierarchy. For example, based on

```
model SomeModel
class SomeClass {
  concept characteristic c :T1
          characteristic d :T2 }
```

some user may define

```
model MyModel
from SomeModel import SomeClass
class MyBaseClass
class SomeClass refines MyBaseClass {
  concept
    characteristic c :T3 ; changed type
    characteristic d unused ; omitted
    characteristic e :T4 } ; new attribute
```

Note that class redefinition has neither subtype semantics nor does is create revisions of types. Instead, each model is checked for consistency separately. Therefore, regardless of class definitions being based on imports, changes like a modified class hierarchy and the omission of attributes (keyword `unused`) are sound when looking at one model alone. Relationships between models (for model personalization etc.) are handled by explicit inter-model relationships established by, e.g., initializations with default or computed values. For example, a class `C` like

```
model BaseModel
class C { concept characteristic i :int }
```

can be changed using the `origin` reference to the original class definition to become:

```
model DerivedModel
from BaseModel import C
class C { concept characteristic i :String
          := Integer.toString(origin.i) }
```

This way one can change the type of an attribute and have the new value computed, e.g., when passing an instance from a `BaseModel` context to one using `DerivedModel`.

This way, one can even change attribute kinds, e.g., lift a characteristic value

```
class Painting {
  concept characteristic painter :String }
```

to a relationship

```
class Painting {
  concept relationship painter :Painter
    :=lookfor Painter{name=origin.painter}}
```

D. *A Model Compiler for Concept-oriented CMSs*

Due to the openness and dynamics requirements CCMSs call for specific implementations. Both well-known extreme software development approaches, individual development and generic software, fail to meet these requirements: individual software is not dynamic since it needs interference of programmers when model changes occur. Generic software does not meet the openness constraint since it prescribes certain model constructs that cannot be overcome and require the user to translate concepts as expressed in the generic language. Therefore, automatic software generation in conjunction with a fine-grained architecture is necessary to allow dynamics of information systems.

There are different approaches to the problem of generating whole software systems which are composed of various parts that are produced by independent generators: (1) the generated software modules have to be adapted in order to be composed [7], (2) generic software modules are wrapped in a domain-specific way [8], (3) glue code to combine modules needs to be generated [9], or (4) the generators need to cooperate in order to create a consistent set of modules. For the fully automatic generation approach required for CCM we favor the latter approach for content management systems.

Writing coordinated generators is a complex task, mainly because setting up an infrastructure for them [10] is difficult. Therefore, our model compiler for content management systems is designed as a framework. In conjunction with a facility for code generation it constitutes a domain-independent meta-programming infrastructure [11].

An instance of the compiler framework is defined by providing a *parser*, one or more *dictionaries*, several *generators*, and a *configuration* of the framework [12].

A typical compiler is divided into frontend and backend [13] in order to decouple source language recognition from target language generation. To this end, a compiler frontend creates an intermediate representation of the input definitions. Such an intermediate representation

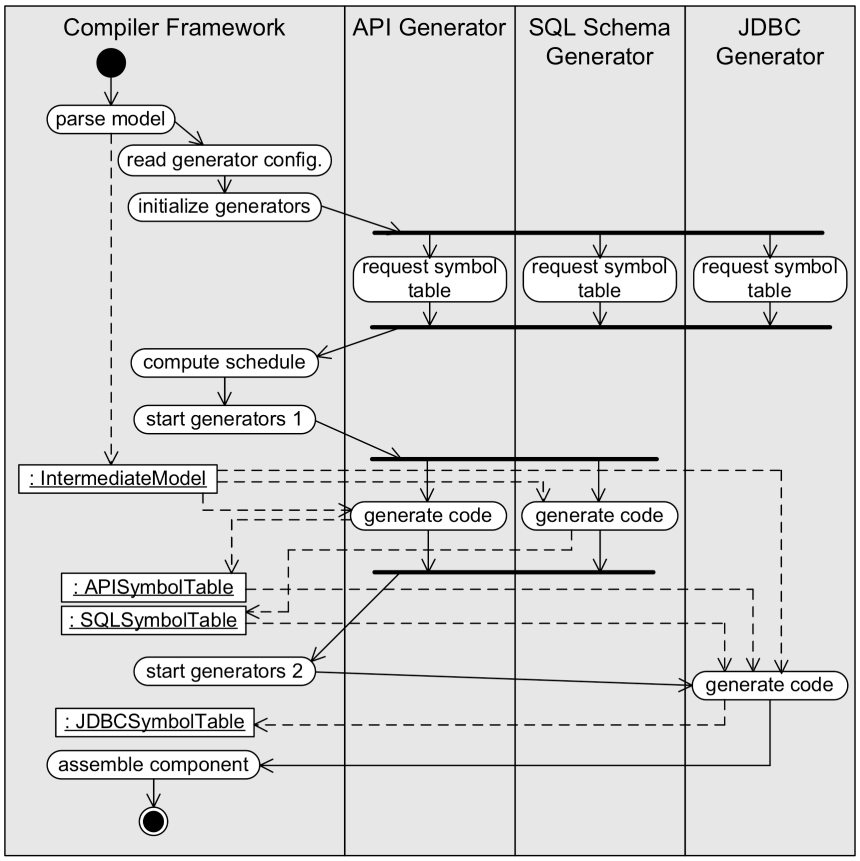


Figure 1. Activity diagram of a sample CCM compiler framework application.

forms the input of a compiler's backend that generates code in the target language. This allows compiler setups for multiple targets as well as – at least in theory – to process different source languages.

The model compiler for our conceptual language is built in an object-oriented fashion. The classical division into frontend and backend has been translated into a framework architecture that allows configuring compilers for the generation of dynamic content management systems. This framework addresses the need to generate multiple targets in conjunction.

A set of parsers is readily available for model compiler instances. The one most commonly used reads files containing asset language expression as defined in Section B. Other options are parsers for different syntactical forms, e.g., in XML, or parsers that adapt an internal model representation from modeling tools. For the purpose of user interface generation, an input language that is related to established presentation technology could be used (see Section V.B).

Alike a programming language compiler that creates an intermediate code representation the frontend in the compiler framework creates *intermediate model* representations in which asset class definitions are available as an object graph.

CCM model compilers have access to one or more dictionaries in which model definitions are stored. This way a compiler gets access to the models named in import statements. It furthermore registers information on the generated code in an own dictionary. This includes the names of implementation classes (e.g., fully qualified Java class names) that have been created for asset classes. Through the dictionaries compilers can create model interrelationships by accessing the information that has been stored by earlier compiler runs.

Code generators constitute the most important extension point of the model compiler framework. Each generator produces one module of a CCMS (see subsequent subsection) in one particular technology. The framework schedules the generators with respect to their dependencies.

There is a direct correspondence between generators and the modules of content management systems. For each implementation of one of the module kinds introduced below there is at least one generator. Often more than one generator contributes to the creation of a module. For example, client modules for database access are typically created by a pair of generators; one of them creates the database schema, the other one creates code to access the database as well as to store and retrieve asset instances.

Sets of generators are given in model compiler *configurations*. Generator instances out of the set of known generator implementations are chosen by means of selecting a configuration. In the context of user interface generation, for example, there are typically different configurations for different presentation technologies used for a CCMS.

Traditional compilers use *symbol tables* to store information about the language constructs recognized. Our model compiler for content management systems builds on the concept of symbol tables, but extends it significantly: these tables are not only used in the frontend of a compiler, but they are the means by which generators communicate during the generation process.

Symbol tables contain detailed information about the artifacts that were created by the respective generator. The aim of symbol tables is to make access to the artifact descriptions explicit for generators that rely on artifacts created by others (and most generators do). Without symbol tables, generators further down the chain would have to make assumptions about naming and would have to recover the corresponding pieces from the whole of the generated artifacts.

Each generator fills its symbol table during its execution and passes the symbol table back to the compiler framework afterwards. The framework in turn gives available symbol tables to further generators making them the essential means of generator communication.

A complete system is normally built from artifacts in several languages. Different meta-programming facilities are available to the generators that share a common intermediate model to create their output.

Figure 1. illustrates the cooperation of generators within the compiler framework. The main task of the frontend is to parse a CCM model definition and to create an intermediate model from it.

As part of the initialization of the generators in the backend, the framework determines the symbol tables each generators needs as input. Based on this information a schedule for generator execution is computed.

The compiler backend passes the CCM model (in the form of an intermediate model) and the required symbol table(s) to each generator. The example shows a setup with three generators. The first one, the API generator, is found in every setup. It creates the uniform module interface with respect to the CCM model. The current implementation creates Java interfaces.

The other two generators together create a client module (s.b.) for use with a relational database management system. One generator creates a relational schema out of the asset model, the other one a module implementation using JDBC to access the database according to the generated schema.

The JDBC generator will always be scheduled last since it requires information on both the schema (to create the proper “embedded” SQL statements) and the module API (in order to make the JDBC module implement it).

The final compilation step is the component assembly. A CCMS component is assembled from the generated modules and parameterizations of third party products when all contributing generators have finished their task. This includes two activities: actually building the modules and combining them in a component of a CCMS.

Modules are built from the generated artifacts. Each generated artifact needs a special final treatment: source code needs to be compiled, database schemata have to be deployed, etc.

E. *An Architecture for Concept-oriented CMSs*

A model-driven code generator – in contrast to programming language compilers – is in full charge of the architecture of the software it generates. This enables the CCM compiler to generate CCMSs in a form that allows incremental compilation. Consequently we have designed an architecture that allows CCMSs to evolve dynamically, thus meeting the dynamics requirement of our content management approach.

The creation of such an evolvable system can in some cases entail changes to its setup. The architecture of the system must therefore allow for flexible reconfiguration. A monolithic system is certainly not capable of such flexible change. Quite the contrary, we propose a modular system architecture that is built of many small modules.

Consequently, the most important concepts of the CCMS architecture are *components* and *modules* [4]. Conceptually, components are units of model reuse, while modules establish code reuse.

Components are logical units that implement one asset model. They are in turn implemented by modules that are each generated specifically for one functionality aspect – like persistence, distribution, transformation, etc. – in one component.

A component is implemented by a combination of modules, usually arranged in layers. Components as software artifacts themselves provide several services to their modules: resolution of identifiers, management of module lifecycles, and initialization of modules at system startup. Each module can use other modules and can also be used by several others. However, the setup of modules in a component always must be a directed acyclic graph.

All modules have a uniform interface and can therefore freely be composed in layers. The module interface reflects the capabilities of the asset language to create, modify, delete and query for asset instances. Each module can thus express its functionality in terms of calls to the module(s) on the underlying layer. This makes it possible to always combine modules in the way most appropriate to the task at hand.

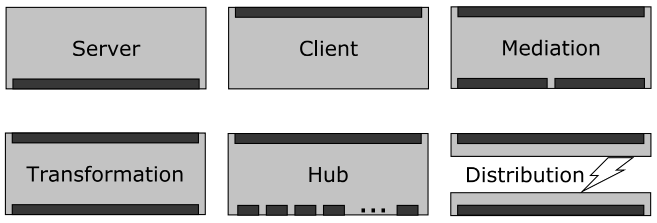


Figure 2. Six kinds of modules of CCMSs.

Figure 2. illustrates the kinds of modules for the most frequently occurring tasks:

- Components are accessed via *server modules* using standard protocols.
- Asset instances (content, characteristic values, and relationships) are stored in third party systems, databases in most cases. *Client modules* perform the mapping of assets from asset models to schemata for such third party systems.

- A central building block of the architecture of generated content management systems is the mediator architecture [14]. Modules of two kinds implement it in our approach. The first are *mediation modules* that delegate requests to other modules based on the request (operation and assets involved).
- The other modules are *transformation modules*. By encapsulating mappings in such modules, rather than integrating this functionality into other modules, mappings can be added dynamically (compare [15]).
- *Hub modules* uniformly distribute calls to a larger number of underlying modules.
- By use of *distribution modules* components can reside at different physical locations and communicate by exchanging data, e.g., XML documents generated from the asset definitions (comparable to the approach of [16]).

These module kinds have been identified with respect to the requirements of content management systems. They provide basic services by the principle of *Separation of Concerns*.

The functionality of a content management system is implemented by a *component configuration* that composes selected modules. Important building blocks are typical module constellations, the perhaps most important one being an implementation of the mediator pattern [14] consisting of a *transformation* and a *mediation module*. Figure 3. illustrates it. Mediator pattern applications are discussed below.

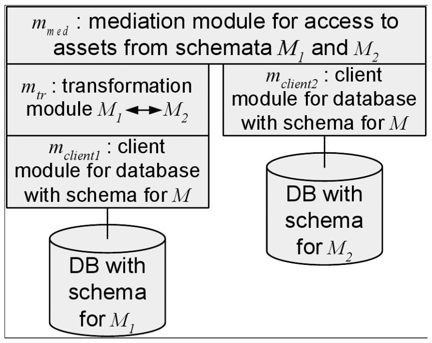

m_{med} : mediation module for access to assets from schemata M_1 and M_2
m_{tr} : transformation module $M_1 \leftrightarrow M_2$
$m_{client2}$: client module for database with schema for M
$m_{client1}$: client module for database with schema for M
DB with schema for M_2
DB with schema for M_1

Figure 3. Architectural building block for evolution in CCMSs

For each kind of module used and for each supported implementation technology there needs to be one generator to equip the compiler framework with.

According to the two ways of combining asset models – model interrelation and personalization – openness and dynamics in CCMSs happen along two dimensions: (1) the *organization* and (2) the *application structure* [17]. Along the organization structure users can define their own views (by personalizing content and schema). Along the application structure, entity descriptions are shared and reused across domains.

In our approach the architecture of the generated systems allows changes along the organization structure by its ability to enable dynamic system evolution and personalization through open redefinition of assets and dynamic invocation of the model compiler [4].

Schema evolution leads to a mediator combination of client, transformation, and mediation modules as indicated in Figure 3. Evolution or personalization requires a mediation module that implements the desired personalization functionality (m_{med} in the figure). Typically this includes the delegation of requests in such way that new instances are created in the component for the new schema (M_2), modifications lead to the creation of a modified copy in that component while removing it from the component holding the outdated model (M_1), and search queries and deletion requests are posed on both components. Such a mediation module can be generated based on the input information, namely a base model and the changes applied to it in a derived model.

Personalization is quite similar, with the difference that modification of an asset leads to the creation of a copy that contains a reference to the personalized asset (instead of deleting the original), and deletion leads to the creation of a *null* asset hiding the original.

The association of models along the application structure is realized by component configurations. Figure 4. shows a configuration that combines two domains – regent and artist descriptions – into the new domain of political iconography. The component is accessed via mediation module m_{med1}. It distributes requests according to the type of the assets on which operations are invoked. If assets from one of the base domains *Regents* or *Artists* are affected, requests are delegated to the mediation module m_{med2}. This mediation module similarly delegates requests further to one of the components holding theses models. These components are accessed via distribution modules $m_{distrib1}$ and $m_{distrib2}$. In the example of Figure 4. the components consist of client modules $m_{client1}$ and $m_{client2}$ and the respective base systems only. Requests to the derived model *Political_Iconography* are forwarded by m_{med1} to the client module m_{client} that manages the users' assets from the political iconography.

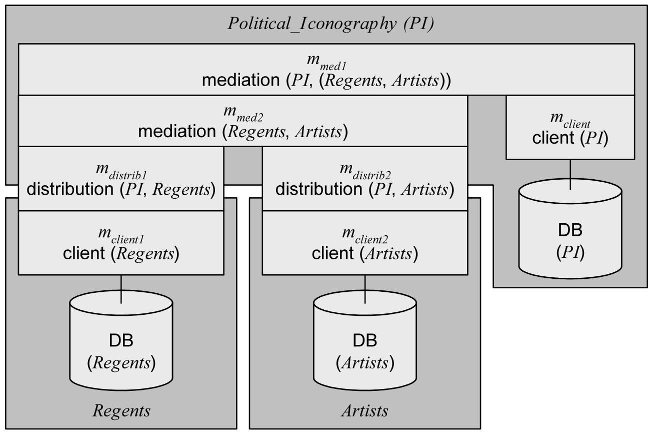

Political_Iconography (PI)
m_{med1} mediation (PI, (Regents, Artists))
m_{med2} mediation (Regents, Artists)
m_{client} client (PI)
$m_{distrib1}$ distribution (PI, Regents)
$m_{distrib2}$ distribution (PI, Artists)
$m_{client1}$ client (Regents)
$m_{client2}$ client (Artists)
DB (PI)
DB (Regents)
DB (Artists)
Regents
Artists

Figure 4. Sample CCMS components for domain interrelation.

As can be seen in Figure 4. the components for *Regents* and *Artists* are integrated into the overall CCMS for Political

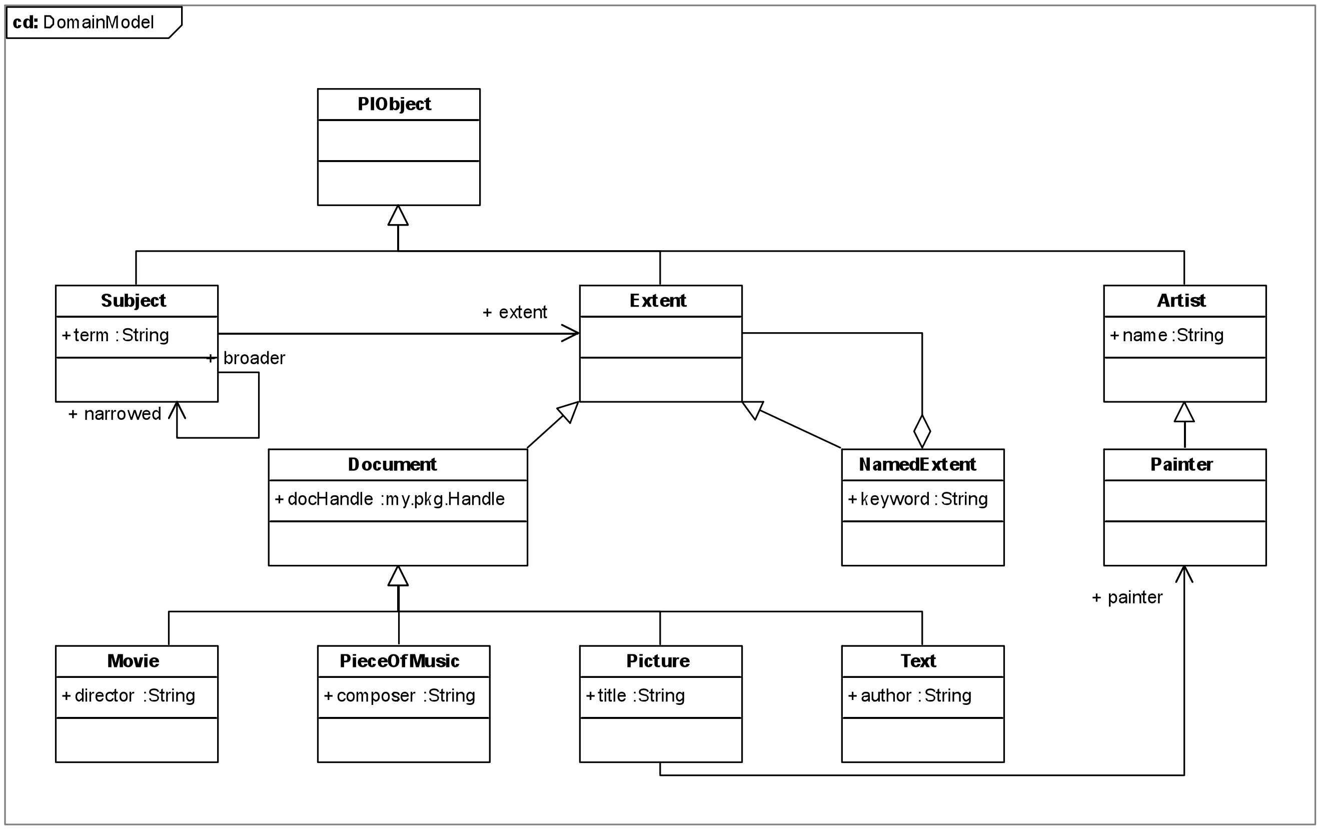


Figure 5. Sketch of a CCM model for the domain of political iconography.

Iconography without modification. This way the components remain unaffected, thus preserving their autonomy, i.e., to be maintained by experts from the respective domains.

III. Concept-oriented Content Modeling

As a running example for the discourse on visualization led in the next section we introduce a tiny asset model in this section. It is a rather condensed extract from a model used in one actual project.

Figure 5. shows an overview in the form of a UML class diagram with attributes for asset characteristics and associations for asset relationships.

A. A Sample Structural Content Definition

The following code shows two simple sample classes:

```
class Document refines Extent {
  content docHandle :my.pkg.Handle }
class Picture refines Document {
  concept characteristic title :String
          relationship painter :Painter }
```

The content handle `docHandle` refers to document data, e.g., a digitized picture for a `Picture` instance. Let the Java class `Handle` be some class to handle references to such data.

The asset class `Picture` describes picture entities like paintings. In inherits the document handle, and defines a conceptual model consisting of a picture's title and a reference to a `Painter` asset.

B. Sample Content Classification by Relationships

Apart from the (structural) definition of the document descriptions it is necessary to define a hierarchy of (semantic) classifiers, here modeled as instances of class `Subject`. The base class `Extent` of `Document` defines the extent of subjects.

These are provided in the form of subject terms and corresponding relationships:

```
class Subject {
 content term :String
 concept
   relationship narrowed :Subject*
   relationship broader  :Subject
     = lookfor Subject {narrowed>={self}}
   relationship extent   :Extent* }
```

The content `term` is the subject term itself, and Java's standard String class is used for instances. The relationship `narrowed` points to more specific subject terms, and `extent` to the documents classified under the term at hand. The reverse of `narrowed`, `broader`, is modeled by a constraint that returns the broader terms based on the (persistent) relationship `narrowed`.

C. A Sample Content Evaluation Rule

The intended form of classification with the sample model given so far is the following: each document is classified under the most specific subject terms that apply. If one wants to use the typical subsumption – documents also showing up under more general terms than those assigned directly – with the definitions made so far this has to be handled by, e.g., the visualization layer.

To make such evaluation rules part of the asset model, additional classes can be introduced for this purpose. For our example we would like to define a special `Subject` with a "deep" extent that takes extents from narrower terms into consideration:

```
class SubjectRec refines Subject {
 concept
  relationship extent :Extent* := {
   origin.extent,
   (create SubjectRec narrowed).extent }}
```

To further stress the importance of a rule-based level, please note that the transitive extent can be expressed by means of relationships (with the help of a productive rule to keep the relationships up-to-date):

```
class SubjectRec2 refines Subject {
  concept constraint deepExtent
            extent >= narrowed.extent
            onviolation modify self {
              extent += narrowed.extent }}
```

This way the recursive extent is materialized in each subject. Of course, the usual problems arise from this redundancy, e.g., updates of extents on picture removal.

IV. Open Dynamic Asset Representations

In this section we shed a light on the second typical task of CMSs, the rendering of representations of content.

The openness and dynamics properties of CCMSs require user interfaces (UIs) to follow model changes. This can only partially be achieved since suitable presentations require manual design [18]; there is no means to automatically produce a visualization that is guaranteed to meet the users' demand for adequateness and ergonomics.

Nevertheless, if visualizations are handled like content, then openness like that of content models can be achieved for them: visualizations can be defined in conjunction with domain models, e.g. group-wise, they can be passed between users that share the same domain models, and then they can be personalized group-wise or individually.

With this kind of user interface modeling presentations are not automatically generated from domain models, but users can define presentations on their own, using a language they are using anyhow. By the correspondence between domain and visualization models the CCM personalization capabilities are beneficial for user interface modeling. Not every user has to define a complete presentation. Usually, domain experts build their models reusing those of other domain experts (by means of personalization) or they use models of neighboring domains (by means of cooperation). Together with the reuse of domain models also accompanying presentation models can be reused.

To this end, we need to avoid "programming" of templates as found in typical CMSs. Instead, declarative definitions of visualization constructs are needed similar to the idea of *Model-Based User Interface Development Environments* [19]. Our aim is to express visualization using the ADL since domain experts already use it (compare [20]), and it allows direct links to domain models. It should be noted that the difference between content and a rendered document is in the eye of the beholder: view classes can be seen as normal classes from the domain of "views".

For the asset-based visualization descriptions we suggest models that follow the Model-View-Controller pattern. In this section the view part of the user interface models is presented. The subsequent section discusses the interrelationship between views and models, as well as the definition of controllers.

Three interrelated models for (1) abstract definitions of presentation components, (2) presentation technologies, and (3) component implementations are provided to CCMS users who can define asset visualizations with the help of these basic contributions for visualization specification.

The models apply to both pure presentations, like web pages, and interactive applications, e.g., for content editing.

Figure 6. gives an overview of the usage of the models presented in the remainder of this paper. The packages *Components* and *Technologies* represent two of the models discussed in this section (elsewhere called *platform model* [21]). The model of component implementations is omitted from the figure since it is not of interest to the domain expert using the models; it is exclusively used by the compiler. The package *Layout* sketches an application of the models (elsewhere called *presentation model* [21]). The relationships to the *DomainModel* are explained in the subsequent section.

A. Presentation Component Model

A very basic contribution for declarative UI descriptions is the *presentation component model*. This model enumerates abstract descriptions of visual components that are usually available in the supported visualization technologies. The following small model excerpt gives an impression of the class definitions, showing a base class of containers for other UI components and a text label class (to display some text):

```
model UIComponents
class UIComponent
class Container refines UIComponent {
  content children :UIComponent* }
class TextLabel refines UIComponent {
  content text :String }
```

Such an abstract component library is used to specify presentations for assets in a platform-independent way:

```
model MyPresentationModel
from UIComponents import ImageLabel,
               Panel, TextField, TextLabel
let picturePanel :Panel := create Panel {
  children := {
    create ImageLabel,
    create Panel {
      children := { create TextLabel,
                    create TextField }
    } } }
```

In this example a user defines a `Panel` consisting of an image, a text label, and a text field. It might be used to display pictures by showing the content (the picture data itself) and its title, where the text label is displaying "Title:" as a label to the text field, and the text field is holding the actual picture's title.

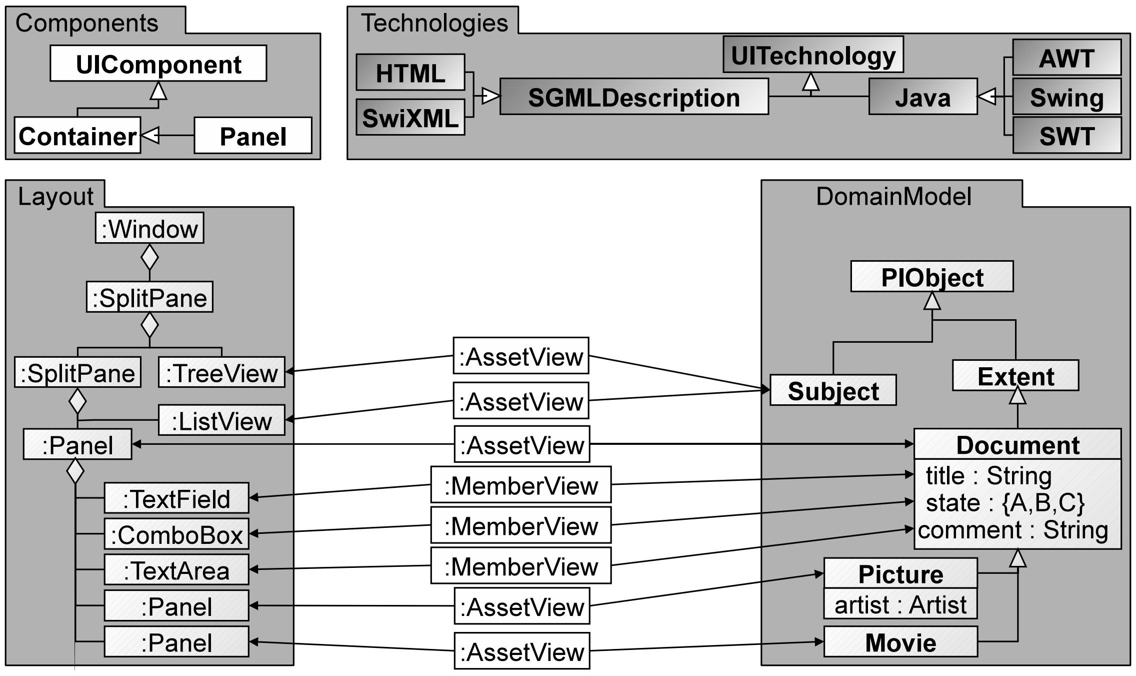


Figure 6. CCM models for user interface component implementations.

B. Presentation Technologies Model

The second basic contribution, the *technologies model*, is rather simple: it just enumerates the supported technologies by defining an asset class for each of them.

A snippet from a technologies model looks like this:

```
model UITechnologies
class UITechnology
class SGMLDescription refines UITechnology
class HTML  refines SGMLDescription
class Java  refines UITechnology
class AWT   refines Java
class Swing refines Java
class SWT   refines Java
```

The sole purpose of these classes is to be referenced in the component implementations model and being passed as a parameter to the presentation generator (see Section D).

C. Presentation Component Implementations Model

The third basic contribution is a model that contains implementations of components in certain technologies. Again a quite small excerpt shall present the basic idea:

```
model UIImplementations
from AssetMetaModel import AssetClass
from UIComponents import Panel,UIComponent
from UITechnologies
  import HTML, Swing, UITechnology
class UIImplementation {
  content prototype :java.lang.Object
  concept relationship component
              :AssetClass < UIComponent
          relationship technology
              :AssetClass < UITechnology }
create UIImplementation {
  prototype := my.HtmlUtils.element("div")
  component := Panel
  technology:= HTML }
create UIImplementation {
  prototype  := new javax.swing.JPanel()
  component  := Panel
  technology := Swing }
```

The model contains one class `UIImplementation` whose instances refer to prototypical implementations (in the current implementation given as Java objects) of abstract component definitions. The set of `UIImplementation` instances defines the pool of implementation artifacts that a UI generator (see Section D) can benefit from.

The link between abstract components and technologies is made by referencing the respective asset classes. The class `AssetClass` is imported from the ADL's metamodel (that is also available in ADL itself) for the required type constraints that furthermore restrict the referable classes to `UIComponent` and `UITechnology`, respectively.

The example sketches implementations of the abstract component `Panel` in HTML (here assuming a helper class to create HTML elements) and in Java Swing.

The definitions in `UIImplementations` are in fact written using more compact statements, but the necessary linguistic means have not been introduced in this paper.

D. Presentation Generation Using Abstract Models

User interface code is generated from the models presented to far, with one addition: links are established between presentation component instances and domain model entities in order to be able to create the demanded adaptive code. The links are based on the `AssetView` and `MemberView` assets as indicated in Figure 6. They will be discussed in Section V.A.

The actual rendering of assets is based on a rather simple algorithm: for each asset class `c` of the domain model to visualize and a technology (from the technologies model) `t`, a UI generator first looks for the UI component(s) to use in the model relating content assets to UI component assets:

```
let v := (lookfor AssetView { type >= c }
        ).view
```

Then implementation prototypes for the UI components can be found in the component implementations model:

```
let p := (lookfor UIImplementation {
           component <= v.type
           technology = t }).prototype
```

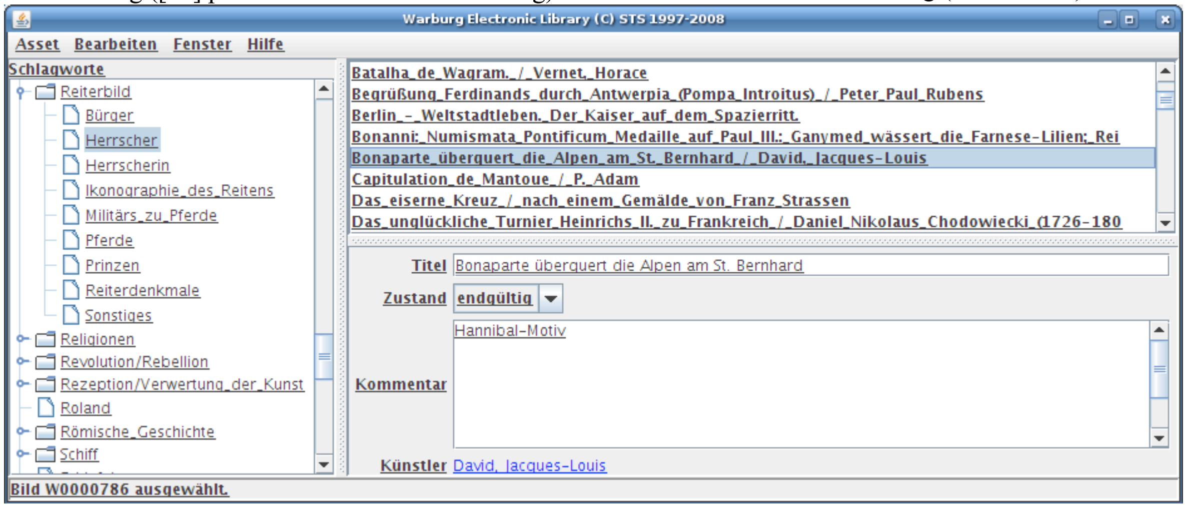
The prototypes are used to create fresh instances that are assembled to a UI as prescribed in the presentation component model. Assembly of the implementations usually has to be performed in a technology-specific way, so that there are specific generators for the supported technologies. These can be easily included in the asset compiler framework (see Section II).

The generated code does not create a static presentation. Instead, the user interface adapts to the asset bound to the presentation (see Section V.A). Such code forms an "adaptation engine" as proposed in [22] and establishes a type-based clustering ([23] presents a time-based clustering).

To this end, the code adjusts the presentation to the bound asset by selecting only those components that are defined for an asset type that matches the current asset's class. The selected components are added to the presentation, and child components that do not apply anymore (because they were added for a previously bound asset) are removed (equivalent to "generation at execution-time" in [24]).

How this adaptation is performed depends on the visualization technology. In Java (AWT, Swing, or SWT) applications, container components can dynamically be altered. Web pages need to be reloaded, or there could be JavaScript code to perform changes dynamically using AJAX (not yet implemented).

Figure 7. shows screenshots of a running Swing application. The screenshots show a GUI with a bound picture asset (a) and a movie asset (b). The panel in the lower right adapts to the bound asset. As can be seen, the artist (*Künstler*) of a picture is given as a relationship to a painter (*Jacques-Louis David*), while the director (*Regisseur*) of a movie is a characteristic string (*Abel Gance*).



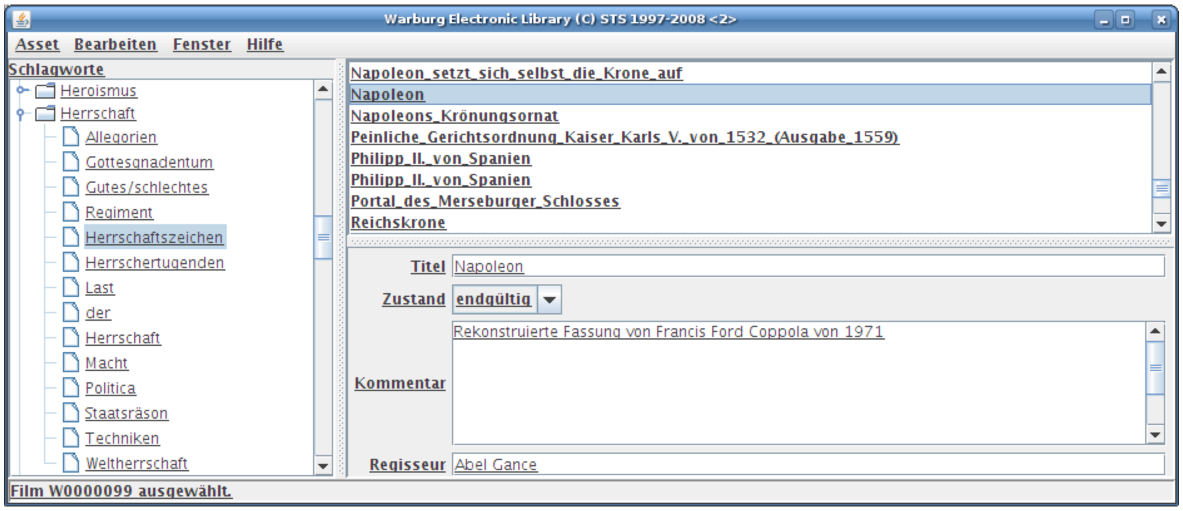
Reiterbild
Bürger
Herrscher
Herrscherin
Ikonographie_des_Reitens
Militärs_zu_Pferde
Pferde
Prinzen
Reiterdenkmale
Sonstiges
Religionen
Revolution/Rebellion
Rezeption/Verwertung_der_Kunst
Roland
Römische_Geschichte
Schiff

Batalha_de_Wagram._/_Vernet,_Horace
Begrüßung_Ferdinands_durch_Antwerpia_(Pompa_Introitus)_/_Peter_Paul_Rubens
Berlin_-_Weltstadtleben._Der_Kaiser_auf_dem_Spazierritt.
Bonanni:_Numismata_Pontificum_Medaille_auf_Paul_III.:_Ganymed_wässert_die_Farnese-Lilien;_Rei
Bonaparte_überquert_die_Alpen_am_St._Bernhard_/_David,_Jacques-Louis
Capitulation_de_Mantoue_/_P._Adam
Das_eiserne_Kreuz_/_nach_einem_Gemälde_von_Franz_Strassen
Das_unglückliche_Turnier_Heinrichs_II._zu_Frankreich_/_Daniel_Nikolaus_Chodowiecki_(1726-180

Titel: Bonaparte überquert die Alpen am St. Bernhard
Zustand: endgültig
Kommentar: Hannibal-Motiv
Künstler: David, Jacques-Louis

Bild W0000786 ausgewählt.

(a) A CCM GUI showing a picture asset.



Heroismus
Herrschaft
Allegorien
Gottesgnadentum
Gutes/schlechtes
Regiment
Herrschaftszeichen
Herrschertugenden
Last
der
Herrschaft
Macht
Politica
Staatsräson
Techniken
Weltherrschaft

Napoleon_setzt_sich_selbst_die_Krone_auf
Napoleon
Napoleons_Krönungsornat
Peinliche_Gerichtsordnung_Kaiser_Karls_V._von_1532_(Ausgabe_1559)
Philipp_II._von_Spanien
Philipp_II._von_Spanien
Portal_des_Merseburger_Schlosses
Reichskrone

Titel: Napoleon
Zustand: endgültig
Kommentar: Rekonstruierte Fassung von Francis Ford Coppola von 1971
Regisseur: Abel Gance

Film W0000099 ausgewählt.

(b) A CCM GUI showing a movie asset.

Figure 7. Screenshots of a generated CCM visualization.

```
model ViewWithLinksToModel
from DomainModel import Picture, Subject
from UIComponents import horizontal, ImageLabel, Movie, Orientation, Panel,
                        TextField, TextLabel, UIComponent, vertical
class LabelAndField refines Panel {
  concept relationship label :UIComponent
          relationship field :UIComponent
          relationship children :UIComponent* = { label, field }
          relationship orientation :Orientation := horizontal }
class DocumentPanel refines Panel {
  concept relationship iconPanel :LabelAndField
          relationship children :UIComponent* := { iconPanel }
          relationship orientation :Orientation := vertical }
class MoviePanel refines DocumentPanel {
  concept relationship directorPanel :LabelAndField
          relationship children :UIComponent* := {super.children, directorPanel}}
class PicturePanel refines Panel {
  concept relationship titlePanel :LabelAndField
          relationship painterPanel :LabelAndField
          relationship children :UIComponent*
                              := { super.children,titlePanel,painterPanel } }
let classifierTree := create TreeView {
  nodeRenderer := create TextLabel }
let extentList := create ListView {
  itemRenderer := create TextLabel }
let moviePanel := create MoviePanel {
  iconPanel     := create LabelAndField {
                     label := create TextLabel { text := "Icon:" }
                     field := create ImageLabel
                   }
  directorPanel := create LabelAndField {
                     label := create TextLabel { text := "Director:" }
                     field := create TextField
                   } }
let picturePanel := create PicturePanel {
  iconPanel    := create LabelAndField {
                    label := create TextLabel { text := "Icon:" }
                    field := create ImageLabel
                  }
  titlePanel   := create LabelAndField {
                    label := create TextLabel { text := "Title:" }
                    field := create TextField
                  }
  painterPanel := create PicturePanel {
                    label := create TextLabel { text := "Painter:" }
                    field := create TextLabel
                  } }
```

Figure 8. Sample layout models for a user interface like the one shown in Figure 7.

V. Model Bindings and Controller Models

In the previous section the static layout of visualizations of assets of specific types has been presented, and it was already specified that the asset presentations should depend on the type of assets bound to the view. This section is concerned about the behavioral aspects of user interfaces. Following the model-view-controller pattern views are related to the two other interface components, models and controllers.

Bindings from the view layout to the domain model are covered by Section A. Section B presents and alternative form of defining layouts and model bindings.

Controllers typically serve one of two purposes: updating the view, e.g., by navigating between assets, and updating the model, e.g., by creating, modifying, or deleting one or

```
model ViewWithLinksToModel
from DomainModel import Movie, Picture, Subject
from AssetUI import AssetView, MemberView
create AssetView  {
  type := Subject           view := classifierTree }
create MemberView {
  member := Subject.term    view :=classifierTree.nodeRenderer }
create AssetView  {
  type := Extent*           view := extentList }
create MemberView {
  member := Document.name   view := extentList.itemRenderer }
create AssetView  {
  type := Picture           view := picturePanel }
create MemberView {
  member:=Picture.docHandle view := picturePanel.iconPanel.field }
create MemberView {
  member := Picture.title   view := picturePanel.titlePanel.field }
create MemberView {
  member := Picture.painter view := picturePanel.painterPanel.field }
create AssetView  {
  type := Movie             view := moviePanel }
create MemberView {
  member := Movie.docHandle view := moviePanel.iconPanel.field }
create MemberView {
  member := Movie.title     view := moviePanel.directorPanel.field }
```

Figure 9. CCM model interrelating views and model classes.

more assets. CCM models to define controllers for view updates are covered by Section C and such to define controllers for model updates in Section D. All kinds of controllers work on regular CCM assets.

A. *Relating Content to Presentation Components*

Users define the presentations they need on the basis of the abstract UI components model. They have to provide two kinds of definitions: an implementation-independent layout description as sketched in Section IV.A and links from content to the UI components that shall display that content.

In easy cases the link between content and a UI component can be made by referring to the content from the content compartment of a UI component. Additionally, the UI component is responsible for the access to the attributes of the asset to be visualized: the selection of members and the decision which relationships to follow (and to which depth). An example for a `Picture` instance `p` would be:

```
model ViewWithValues
from UIComponents import ImageLabel,
                Panel, TextLabel, vertical
create Panel {
  children := {
   create ImageLabel {image:=p.docHandle},
   create TextLabel  {text :=p.title} }
  orientation := vertical }
```

This way of linking content to UI components allows explicitly choosing the presentation for an asset instance, but it requires a complete definition of one instance per content type and desired visualization, without code reuse through classes. Compared to conventional implementations this is typical for simple manually programmed GUIs or such created with the help of interface design tools.

Some reuse can be achieved by defining UI classes for specific content (types) that are instantiated for a matching content instance. This is what typical template languages do.

A higher degree of reuse can be achieved by defining *rules* for the linkage of content to UIs.

The basis for such user-based visualization descriptions is a fourth basic model that defines class-based relationships:

```
model AssetUI
from AssetMetaModel
  import AssetClass, Member
from UIComponents import UIComponent
class AssetView {
  concept relationship type :AssetClass
          relationship view :UIComponent }
class MemberView {
  concept relationship member :Member
          relationship view :UIComponent }
```

The asset class `Member` is defined in the ADL's meta model like the metaclass `AssetClass` is.

Such a basic model can be used for definitions like the views shown in Figure 8. and the relationships shown in Figure 9. according to the graphical sketch in Figure 6.

The example shows a small excerpt of a model that defines a GUI like that from Figure 7. It consists of a tree showing the `Subject` hierarchy, the `Extent` list of one `Subject`, and a panel with the selected `Extent`.

Standard components are used for the tree and for the list in the example. These are configured with one component to render tree nodes and list items, respectively.

For the `Extent` assets one Panel per type is defined (the example of Figure 8. shows just excerpts of the panels for `Movie` and `Picture` instances). To be able to correctly set the horizontal orientation of, e.g., labels and text fields, and the vertical orientations of the components for the attributes, a helper class `LabelAndField` is defined. It allows to define orientations at the class level.

As sketched in Section IV.D, view components have to be related to domain model elements. Using the `AssetUI` model sketched above, the instantiations of `AssetView` and `MemberView` prescribe the rendering of all assets of a user-defined type. In the example of Figure 9. `Picture` (and subtypes) instances are defined to be rendered by a picture view, and values of their title attributes are rendered in a text field. (Expressions like `Picture.title` return the `Member` instance describing the named member.)

As can be seen in the example, a dedicated `Panel` class for `Picture` instances is defined. It is important to note that a relationship between *the class* `Picture` – not a `Picture` *instance* – and the instance `picturePanel` with its child components is established.

Whenever an asset is to be visualized, a suitable UI component can be found depending on its type as shown in Section IV.D. The component implementation instance is used in two ways: the first use is by a UI generator that uses it as a pattern for the generation of code that creates a component implementation at runtime. Examples are Java code that produces a rich Swing client or a JSP page that incorporates HTML fragments.

The second use is the running code itself that adapts a UI to a new or changed asset instance that is to be visualized by it. To this end, the information from the layout model and the links to the domain model are included in the generated code.

B. *An Alternative Asset Language for Web Presentations*

As indicated in Section II.D the CCM compiler framework allows using custom parsers for specific syntactic forms of model definitions. We currently investigate the use of HTML for layout definitions with embedded tags for the relationship to domain assets.

Specifying user interfaces by HTML with embedded tags allows using web design tools (as long as they leave the custom tags intact). Though this violates the idea of dynamics to some extent, it is important in projects where web designers create visualizations for users of a domain.

There are two custom tags that allow to express the `AssetView` and `MemberView` relationships. The semantics of the `<assetview>` tag is the following: if the asset currently to be displayed is of the type given by the type attribute, then the content of that tag is rendered; otherwise, it is excluded from the page.

The `<memberview>` tag is evaluated to the asset's member given by the `name` attribute. Following the example of the *JavaServer Pages Standard Tag Library* (*JSTL*) it optionally allows to define a variable. Then the tag is not expanded to the member's value, but instead the named variable is initialized with it. Later on the variable can be referenced by using the *Expression Language* (*EL*) of JavaServer Pages.

A page definition using this language might look like in the following example:

```
<html>
  …
  <ccmui:assetview type="Document"><table>
    <tr>
      <th>Icon</th>
      <td><ccmui:memberview
        var="icnsrc"
        name="docHandle"
        format="url"
        /><img src="${icnsrc}"></td>
    </tr>
    <ccmui:assetview type="Picture"><tr>
      <th>Title</th>
      <td><ccmui:memberview name="title"
        /></td>
    </tr></ccmui:assetview>
  </table></ccmui:assetview>
  …
</html>
```

In this example a table is rendered for `Document` instances. If the current asset is actually a `Picture`, then the table has two rows, one for the image (in this example, `docHandle` is supposed to always refer to an image file) and one for the `title`. For all other document instances (e.g., movies) the table contains only the row with the image content.

Of course, users can still alter such enriched HTML layouts since they are processed by a CCM compiler. But this requires users to have knowledge on HTML as well as on the custom tags.

C. *View Controllers*

There are interactive view elements that update other views, in particular by navigating from one asset to another. View updates can be formulated using the ADL by defining constraints on the view assets.

The following definitions establish synchronization between the subject tree and the extent list in the sample client shown in Figure 7.

```
class TreeListSynchronizer {
  concept
    relationship tree :TreeView
    relationship list :ListView
    constraint listInSyncWithTree
     (tree.selection=na and list.model=na)
      or (tree.selection # na
          and tree.selection.extent
              = list.model)
      onviolation modify list {
        model := tree.selection.extent } }
create TreeListSynchronizer {
  tree := classifierTree
  list := extentList }
```

In this example we use a constraint on the subject tree and the extent list. We define a class for the pair of them and create an instance so that the constraint is active. A `TreeView` has a `selection` relationship holding the currently selected tree node. If no node is selected (`na` as the null value for no asset) then the list should be empty, meaning its model does not refer to an asset set. Otherwise, we require the model of the extent list to be equal to the extent of the selected subject.

When this constraint is violated the repair code in the `onviolation` clause assigns the list model according to the definition. Note that in case of an empty selection the expression in the `modify` statement results to `na`, and the list model is thus correctly cleared.

D. Controllers to Manipulate Models

Since both the domain model and the view model are formulated using the ADL, no specific technology is required to alter domain assets from within a user interface. The usual asset manipulation language commands can be used to modify assets from the domain model, and the CCM model compiler will create suitable target code from these commands.

To trigger commands on the domain model these can be wrapped in `Action` assets. `Action` is a predefined user interface class that has one relationship `perform`. This relationship can be defined with a constraint to form a kind of function. Instances are used at runtime by interactive components: these can be assigned an `Action` asset, and generated code will use the relationship `perform` to trigger the defined command that can, as a side-effect, modify assets from the domain model.

The following class definition gives an example for an action to store the modifications applied to a `Document` currently visible in the document panel. Let `docPanel` be a reference to the currently used document panel in the client, e.g., `picturePanel` in the case of a `Picture`:

```
class CommitAction refines Action {
  concept
   relationship perform :Asset*
     = modify docPanel.model {
         title := titlePanel.field.text
         …
       } }
```

When the `Action` from the example is related to an interactive UI component, e.g., a menu item, the `modify` statement on the `docPanel`'s model will be executed when interaction takes place (by dereferencing the `Actions`'s `perform` relationship). It updates the currently bound asset in that way that it assigns the updated values from the input fields to the asset's attributes.

An interactive UI component can be a standard component, e.g., a menu item or a button as well as a complex component [25]. E.g., a document panel as sketched above may trigger actions if any of the enclosed text fields has its value changed.

VI. Outlook

The first research direction that needs attention is a higher level of abstraction for the view component model. The initial design was targeted at fat clients and web pages. For these kinds of user interface approaches the presented view model is suitable. But new platforms arise, the most important one being mobile devices, but also interactive whiteboards, tables, etc. For some of these platforms there is no one-to-one mapping from logical view components to component implementations. Instead, some components are realized in a completely different way as they are on conventional graphical user interfaces.

To target such devices more abstract view models are required, and an additional processing step has to create a concrete view for a specific device. Only then the simple construction algorithm based on prototypes as presented in this paper can be applied.

The additional processing step can be realized by model-to-model transformations [26] that generate target asset models from source asset models. The CCM compiler framework can be used for such a model-driven software development approach. Generators that realize device-specific presentation patterns can process source models with more abstract view definitions and create more concrete view models for, e.g., device-specific layouts.

A second major research topic is that of UIs incorporating more than one technology. In practice, such hybrid definitions are regularly used, e.g., web presentations often use a mixture of layout descriptions and program code, like HMTL embedding Java that in turn embeds SQL, or the current trend to enrich web pages with flash animations and JavaScript code, eventually forming AJAX or Flex clients. Yet, a theoretic foundation for such hybrid language approaches is largely missing.

The aim of the CCM approach is to enable domain experts to create models on their own regardless of software development constraints. Currently they do so by using the ADL to formally define their information needs. But for many users the presentation level is the means to argue about models. One goal is to allow users to change presentations in order to express the demand for domain model changes or personalization, and to analyze the changes in order to derive the appropriate domain model changes. Though this should be undecidable for general changes, it might be tractable to recognize certain patterns. This approach would lead to some kind of agile user-centric design approach.

Acknowledgment

The author thanks Joachim W. Schmidt for countless discussions on the topic *CCM* and the seemingly never-ending energy still going into it. Furthermore we appreciate the cooperation with numerous project partners. In particular, we want to thank our colleagues from art history for sharing their insights into the description and management of multimedia artifacts with us. Finally, the author thanks his employer, the T-Systems Multimedia Solutions GmbH, for the opportunity to follow his scientific ambitions.

References

[1] Hans-Werner Sehring, “Adaptive Content Visualization in Concept-oriented Content Management Systems,” Proceedings of the First International Conference on Creative Content Technologies, Athens, Greece, 2009.

[2] Gerd Kamp, “Multichannel publishing,” OBJEKTspektrum, 2001.

[3] Gustavo Rossi, Daniel Schwabe, and Robson Guimarães, “Designing personalized web applications,” Proc. World Wide Web 2001, ACM Press, 2001, pp. 275-284

[4] Hans-Werner Sehring and Joachim W. Schmidt, “Beyond Databases: An Asset Language for Conceptual Content Management,” Proceedings of the 8th East European Conference on Advances in Databases and Information Systems, LNCS, vol. 3255, Springer-Verlag, 2004, pp. 99-112

[5] Ernst Cassirer, Language, Mythical Thought, and The Phenomenology of Knowledge. Vol. 1-3, The Philosophy of Symbolic Forms, Yale University Press, 1965.

[6] C.S. Peirce, Collected Papers of Charles Sanders Peirce. Harvard University Press, Cambridge, 1931.

[7] Johan Brichau, “Integrative Composition of Program Generators,” PhD thesis, Vakgroep Informatica, Vrije Universiteit Brussel, 2005

[8] Gopal Gupta, “A Language-centric Approach to Software Engineering: Domain Specific Languages Meet Software Components,” Electronic Proceedings of the CoLogNet Area Workshop Series on Component-based Software Development and Implementation Technology for Computational Logic Systems, 2002

[9] Uwe Assmann, “Meta-programming Composers In Second-Generation Component Systems,” J. Bishop and N. Horspool, Systems Implementation 2000 – Working Conference IFIP WG 2.4, Chapman and Hall, 1998

[10] Yannis Smaragdakis and Don Batory, “Scoping Constructs for Program Generators,” technical report, no. CS-TR-96-37, University of Texas at Austin, 1996

[11] Yannis Smaragdakis, Shan Shan Huang, and David Zook, “Program generators and the tools to make them,” PEPM~'04: Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, ACM Press, 2004, pp. 92-100

[12] Hans-Werner Sehring, Sebastian Bossung, and Joachim W. Schmidt, “Content is capricious: a case for dynamic system generation,” Proceedings of the 10th East European Conference on Advances in Databases and Information Systems, LNCS, vol. 4152, Springer-Verlag, 2006, pp. 430-445

[13] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, “Compilers: Principles, Techniques, and Tools,” Addison-Wesley, 1986

[14] G. Wiederhold, “Mediators in the architecture of future information systems,” IEEE Comp., vol. 25, 1992, pp. 38-49

[15] Mira Mezini, Linda Seiter, and Karl Lieberherr, “Component integration with pluggable composite adapters,” Software Architectures and Component Technology, Kluwer, 2000

[16] German Shegalov, Michael Gillmann, and Gerhard Weikum, “XML-enabled work-flow management for e-services across heterogeneous platforms,” VLDB Journal, vol. 10, no. 1, 2001, pp. 91-103

[17] Hans-Werner Sehring, “Konzeptorientiertes Content Management: Modell, Systemarchitektur und Prototypen,” dissertation (in German), Hamburg University of Science and Technology (TUHH), 2004

[18] Pedro J. Molina, “A Review to Model-Based User Interface Development Technology,” Hallvard Trætteberg, Pedro J. Molina, and Nuno Jardim Nunes, "MBUI 2004, Making model-based user interface design practical: usable and open methods and tools, Proceedings of the First International Workshop on Making model-based user interface design practical: usable and open methods and tools, CEUR Workshop Proceedings, volume 103, 2004

[19] Paulo Pinheiro Da Silva, “User Interface Declarative Models and Development Environments: A Survey,” Proceedings of DSV-IS2000, LNCS, vol. 1946, Springer-Verlag, 2000, pp. 207-226

[20] Jürgen Falb, Roman Popp, Thomas Röck, Helmut Jelinek, Edin Arnautovic, and Hermann Kaindl, “Fully-automatic generation of user interfaces for multiple devices from a high-level model based on communicative acts,” HICSS '07: Proceedings of the 40th Annual Hawaii International Conference on System Sciences, IEEE Computer Society, 2007

[21] Jacob Eisenstein, Jean Vanderdonckt, and Angel Puerta, “Applying Model-Based Techniques to the Development of UIs for Mobile Computers,” Proceedings of the 6th international conference on Intelligent user interfaces, Santa Fe, New Mexico, United States, ACM, New York, NY, USA, 2001, pp. 69-76

[22] Steffen Lohmann, J. Wolfgang Kaltz, and Jürgen Ziegler, “Model-driven dynamic generation of context-adaptive web user interfaces,” Models in Software Engineering, LNCS, vol. 4364, 2007, pp. 116-125

[23] Mittapally Kumara Swamy and Polepalli Krishna Reddy, “An Efficient Context-Based User Interface by Exploiting Temporality of Attributes,” APSEC '09: Proceedings of the 2009 16th Asia-Pacific Software Engineering Conference, IEEE Computer Society, 2009, pp. 205-212

[24] Windson Viana and Rossana M. C. Andrade, “XMobile: A MB-UID environment for semi-automatic generation of adaptive applications for mobile devices,” Journal of Systems and Software, vol. 81, no. 3, Elsevier Science Inc., 2008, pp. 382-394

[25] Sébastien Romitti, Charles Santoni, and Philippe François, “A design methodology and a prototyping tool dedicated to adaptive interface generation,” Proceedings of the 3rd ERCIM Workshop, 1997

[26] Tom Mens, Krzysztof Czarnecki, and Pieter Van Gorp, “A Taxonomy of Model Transformations,” Jean Bezivin and Reiko Heckel, Language Engineering for Model-Driven Software Development, Dagstuhl Seminar Proceedings, no. 04101, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005