# Conceptual Content Modeling and Management
## The Rationale of an Asset Language

Joachim W. Schmidt and Hans-Werner Sehring

Technical University Hamburg-Harburg, Software Systems Department,
Harburger Schloßstraße 20, D-21073 Hamburg, Germany

{j.w.schmidt, hw.sehring}@tu-harburg.de

**Abstract.** Focused views on *entities of interest* – concrete or abstract ones – are often represented by texts, images, speech or other media. Such media views are communicated through visual or audio channels and stored persistently in appropriate containers.
In this paper we extend a computational content-container model into a closely coupled content-concept model intended to capture more of the meaning – and improve the value – of content. Integrated content-concept views on entities are modeled by the notion of *assets*, and our asset language aims at two goals derived from extensive experiences with entity modeling:

1. *Expressiveness*: according to Peirce [29] and others, entity modeling – and, therefore, also asset modeling – has to cover three different perspectives:
   - an entity's inherent *characteristics* (firstness categories);
   - its *relationships* to other entities (secondness categories);
   - the *systematics* behind the first two perspectives (thirdness categories).
2. *Responsiveness*: according to Cassirer [8, 47] and others, entity modeling *processes*, in order to be successful have to be
   - *open*, i.e., users of an asset language must be able to adapt their asset models according to the requirements of the entity at hand;
   - *dynamic* in the sense that all aspects of an asset model must be subject to inspection and adaptation at any time.

Our current experiments with asset languages are motivated by the need for a better understanding and integration of content *and* concepts about application entities. We conclude by outlining a component-based implementation technology for open and dynamic asset systems.

## 1 Introduction: Motivation and Rationale

The management of structured data is well understood in computer science and so is their use in software system engineering [21, 46, 45]. Data are essentially maintained as content of typed variables or schema-constrained databases, and data access and manipulation is abstracted by functions or transactions and encapsulated by software components.

The object-oriented approach, for example, aims at a seamless support of software development from application analysis through system design to software implementation and provides a methodological basis as well as tool support for the realization of large-scale object systems.

A rapidly increasing class of computer applications use persistent object systems as containers for any kind of (multi-media) content. Such applications face the problem that the computational models by which the container objects are defined say nothing about the application concepts associated with their content. As a consequence, such systems can give only little and incoherent support for content retrieval, presentation, change, explanation, etc.

In this paper we argue that application contents *and* application concepts need to be closely coupled and represented by a single notion which we call an *asset*. Assets represent intimately allied content-concept pairs which represent and signify application entities. The content aspect of an asset holds a media view on an entity while the concept aspect represents its allied concept view.

In section 2 of this paper we first introduce objects as providers of container functionality and then define assets on top of container objects where the concept aspect of an asset is intended to model application aspects of entities and is not restricted to computational purposes. Section 3 discusses the *expressiveness* of an asset language and section 4 outlines such a language. In section 5 the modalities by which an asset language is made available to its users are discussed under the heading of language *responsiveness*. Both asset language expressiveness and responsiveness put particular demands on asset system implementations. Section 6 summarizes aspects of an asset technology. The paper concludes by reporting on ongoing interdisciplinary research and development projects and refers to commercial activities.

In summary, our experiments with asset languages and systems serve essentially two goals:

– advancing large-scale content management by a concept-based view on application content and, vice versa,
– improving the representation of application concepts by means of a generalized notion of content.

## 2 Adding Meaning to Content

Object-oriented models and technology are highly appropriate in modeling digital containers for various kinds of application content and their presentation media. However, the *meaning* of content, although captured in early phases of object-oriented application analysis and software design, cannot be represented adequately by object-oriented computer languages. Therefore, we propose an asset language capable of representing both computational concepts for containers as well as application concepts for their content.

## 2.1 Computational Objects as Containers for Application Content

Since the early days of high-level programming the states of computations are modeled by typed variables which hold values from predefined value sets and which can be accessed by predefined operations:

```
var contents: Integer := 0;
```

Here a computational object of variable content is associated with the mathematical concept of integer numbers and is initialized by the specific content *zero*. The specification of high-level programming languages regulates the necessary details of how to use computational objects: their lifetime and visibility or their access from right- and lefthand-side expressions. Since algorithmic programming takes a mathematical view on its application domains the application entities *are* numbers or Booleans and, therefore, there is a perfect match between content – e.g., the mathematical entity *zero* – and the associated mathematical concept of `Integer`.

In software engineering scenarios where re-usability of designs, code and content is a central issue, computational objects are modeled more explicitly. By an object-based language, for example, we can hide the above integer variable and define integer cells with their own methods and possibly additional properties [1]:

```
object myIntegerCell {
  var contents: Integer := 0;
  method get(): Integer {return self.contents};
  method set(n: Integer) {self.contents := n}; }
```

The cell type, `IntegerCellType`, is defined by the signature of the above cell object, `myIntegerCell`. Cells of this kind are perfectly suited for algorithmic computing, i.e., for any application where cell content is modeled by mathematical or logical concepts which are used for computation only.

Nowadays, however, the majority of applications come from other areas – business enterprises, e-commerce, public administration, logistics, art history etc. – and application entities have to be mapped from their own conceptual context into the computational concepts of a computer language. Object-oriented analysis and design (OOAD) processes support such mapping processes which may result in an `imageContainer` object definition such as

```
object myImageContainer {
  var contents: array of byte := emptyImage;
  method get(): array of byte {return self.contents};
  method set(i: array of byte) {self.contents := i}; }
```

or, for class-based languages [1], in a corresponding class definition, `ImageContainer`. Classes support a more dynamic object generation by

```
myImageContainer := new ImageContainer; ...;
```

Classes also can be re-used for the definition, for example, of specialized containers such as

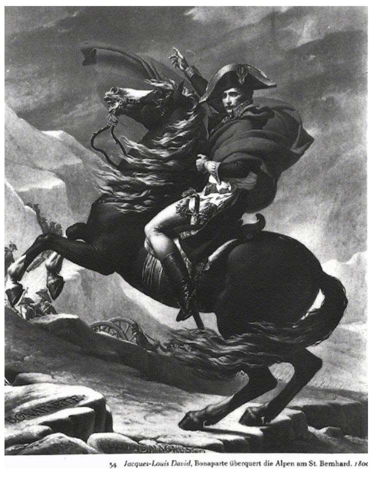
BONAPARTE
54 Jacques-Louis David, Bonaparte überquert die Alpen am St. Bernhard. 180

**Fig. 1.** Preview thumbnail computed from `myImageContainer.contents`

```
class JPEGContainer refines ImageContainer {
  var ...; ...;
  method getThumbnail(...): SmallJPEGImageCopy
    {... self.contents ...};
    //computes thumbnails from JPEG content and its parameters;
    //may be used for preview etc., see fig. 1
  method getColorDepth(): Integer {...};
  method getCompressionRate(): Integer {...}; }

myJPEGContainer := new JPEGContainer; ...;
```

The above example sketches a class which interprets the byte contents in a specific way. Furthermore, it introduces a method `getThumbnail()` which returns a thumbnail version of the object's contents. It is important to note that the contents of computational objects such as `myJPEGContainer` very often serve two purposes:

– *computational* use, e.g., to compute thumbnails from a container's digital content;
– *conceptual* use, e.g., to be interpreted by humans as a thumbnail of some *equestrian statue* (fig. 1).

Some conceptual model [6] of the notion of *equestrian statue* may have existed as an OOAD document [21] in earlier phases of a software engineering project on iconographic digital libraries but is no more available at runtime.

Consequently, we argue for a language with modeling expressiveness and runtime responsiveness by which we can serve both computational purposes as required for an efficient container technology as well as conceptual purposes as requested for the deeper understanding of the application content maintained by such containers and of the concepts associated with it.

### 2.2 Content-Concept Pairs as Assets for Entities

As learned from database design and from application software engineering in general, data – or content in the above sense – does not come *out of the blue* but is a result of a careful conceptualization process of the *entities of interest* in some application domain.

In our approach we complement the *media view* on an application entity (as represented by content, see fig. 1) by a *model view* represented by named and related concepts associated with entities. Content-concept pairs – which we call *assets* (see fig. 2) – can be defined, queried and manipulated as elements of our *asset language* (see section 4).

In our example domain of *Political Iconography* some person on horseback may be considered as a figure of political relevance and iconographic effect. Such entity may be represented by an asset instance with an asset identifier, say, `AID1`. The content aspect of asset `AID1` may contain an image of the entity at hand as represented by fig. 1. The concept aspect describes the entity by its *characteristics* (firstness) and by *relationships* to other entities (secondness), for example, `AID2` and `AID3`. Our example asset is defined as an instance of an asset class (thirdness), *EquestrianStatue*, which regulates to some extent the *systematics* behind the bindings to (1) and (2).

A first sketch of asset instance `AID1` may look as follows:

```
AID1 → [content {see fig. 1} | concept sex    male (1) ...
                                       photo  ♘    (1) ...
                                       artist AID2 (2) ...
                                       ruler  AID3 (2) ...]
                                        EquestrianStatue (3)
```

Our Asset Definition Language is discussed in some detail in section 4. It is based on experiences from entity modeling in database design and software systems engineering.

## 3 On Expressive Entity Modeling

According to Peirce [29] and others (see, for example, Sowa [39]), *expressive* entity modeling has to cover an entity from three different perspectives:

1. inherent *characteristics* of an entity (Peirce firstness categories);
2. *relationships* between an entity and other entities (Peirce secondness categories);
3. *systematics* of entity *genesis* as provided by the "business procedures" behind legal bindings (Peirce thirdness categories) for perspectives 1 and 2.

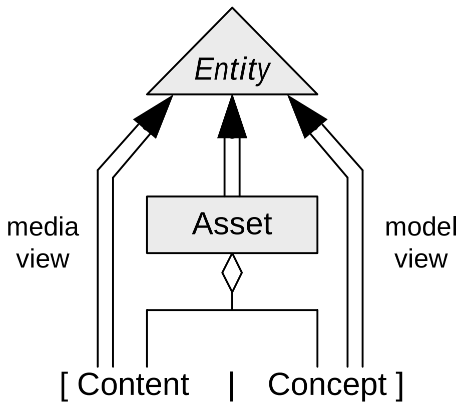

**Fig. 2.** Entities and Assets, Overview

As an example for the three perspectives take a human who is modeled as *Person* by the firstness characteristics, age, sex, nationality etc. and as *Employee* by additional secondness relationships to other entities such as his head of division, task assignments, past employments etc. Description on thirdness level would further specialize the employee as an *EmployeeOfCompanyABC*, for example, by ABC's specific business procedures for hiring, firing or promoting its personnel or for task assignment (see also section 3.3).

Approaches such as entity-relationship modeling for database design [10] cover essentially the firstness and secondness perspectives. Numerous ER extensions (HERM [45], ..., UML [46]) contribute to the thirdness dimension by various kinds of constraints (invariants, type constraints, pre- and postconditions etc.).

As equally important as the expressiveness of an entity model are the modalities by which such expressive models can be utilized, i.e., their *responsiveness.* Cassirer in his work on *Symbolic Forms* [8, 47] studies at length the closely coupled roles of content *and* concept for human domain understanding and emphasizes the *open-dynamic* character of content-concept use. Whitehead, as another example, also states quite clearly, "We must be systematic, but we should keep our systems open" [39]. Responsiveness issues are further discussed in section 5.

To summarize, the rationale behind our content-concept-based asset approach is the assumption that entities are captured by providing both a media view based on content and a model view based on concepts. Our asset language combines computational content management as enabled by advanced object-oriented container technology with insights from the Peirce and Cassirer programs for expressive and responsive conceptual modeling.

### 3.1 Entity Characteristics

On the level of firstness categories, assets are defined by a choice of *characteristic* properties inherent to an entity. They are modeled by the asset language's base types – `String`, `Integer`, `Boolean` etc. as well as structures over it – or by computational objects as, for example, for `Date` or `SmallJPEGImageCopy`.

```
asset someEquestrianStatue {
  content image: JPEGContainer;
  concept characteristic sex: {female, male};
          characteristic paintedAt: Date;
          characteristic title: String;
          characteristic photo: SmallJPEGImageCopy;...}
```

Standard methods such as get, set, etc. need not be defined explicitly but are generated by the asset language compiler together with standard forms for method invocation and parameter binding (see section 6.1).

### 3.2 Entity Relationships

Asset *relationships* model an entity on the level of secondness, i.e., by its association to or interaction with other entities [38]. Since entities are represented by assets, an asset relationship is essentially a named binding of the asset at hand – the *source* asset – to other assets – the *target* assets. Target asset bindings are initialized by defaults and controlled by target asset classes and their constraints.

```
asset someEquestrianStatue {
  content ...
  concept characteristic ...
          relationship paintedBy: Artist;
          relationship requestBy: Ruler;
          relationship depicted:  Person; ... }
```

### 3.3 On the Systematics of Entity Genesis

In all mediating organizations, managed companies, organized domains, etc. there are rules by which concrete characteristics and relationships of entities is systematically regulated to some degree. During the analysis and design phase of computerized support systems such systematics of entity genesis – the entity definition on the Peirce thirdness level – are captured by various (semi-)formal approaches. System implementation maps the resulting sets of predicates into conditionals on the software execution level:

- object-oriented software engineering extracts such regulations from textual documents on business procedures and maps them into object designs via semi-formal use-case diagrams [21];
- formal program specification captures program semantics, for example, by invariants as well as pre- and postconditions and applies predicate transformers for code development [19];

– workflow systems have their own pragmatic workflow specification and enactment languages [49], and
– database systems enrich their schema definitions by constraints and triggers which restrict transactions and thus protect database content.

In all cases, thirdness level entity modeling is concerned with the semantics of the set of procedures which handle entities and, therefore, create and change entity descriptions. We concentrate on the use of types and constraints for thirdness level entity modeling and experiment with assets on various levels:

– asset language level: asset type and class definitions as well as constraints on asset characteristics and relationships (*intensional* semantics, see section 4.3);
– asset instance level: *extensional* asset class semantics by representative collections of asset instances; exploited, for example, as training sets for automatic asset classification or for learning scenarios (see section 4.3);
– asset system level: API signatures for process and workflow bindings.

In our approach thirdness issues usually remain underspecified in the sense that details such as binding orders etc. are left open. In an asset's content part there is, however, ample room for all kinds of semi-formal working documents on constraint and procedure specification.

In practice, the firstness and secondness aspects of assets are connected by their use as initialization and manipulation parameters of thirdness "business processes". Looking at assets from such a work perspective we can take a dual position [32]:

– assets for *works to be* which we can bind to, inspect, evaluate, etc.
– assets for *work to do* which we can initialize, put forward, interrupt, resume, refine, undo, redo, etc.

## 4 Preview of an Asset Definition Language

As with database management, the functionality of an Asset Languages can be subdivided into three subareas: Asset Definition (ADL), Manipulation (AML) and Querying Languages (AQL). In this paper asset definitions are expressed in a linguistic form – by the expressions of our Asset Definition Language – while asset manipulation and querying is performed via interactive, form-oriented interfaces generated from ADL expressions.

### 4.1 Content Modeling: Asset Container Objects

One of the two indivisible sides of an "asset coin" is its (reference to) content. The content aggregated in an asset is defined in its content part and the entire power of object-oriented container technology is employed here (see section 3):

```
class EquestrianStatue {
  content image: JPEGContainer;
  concept ... }
```

Details of content storage and retrieval regulate, for example,

- **content storage location:** an object store may contain the entire content as indicated in the examples from section 2.1 (`byte array`), or just a reference to externally stored content. In the latter case the container provides (transparent) access to the actual content. In addition issues like caching (for repeating access), archiving (for content vanishing from an external system), mediation (to transparently access more than one external system), etc. may be handled.
- **content delivery mode:** content may be delivered en bloc (as in the case of the byte array) or streamed, QoS parameters [40] might apply, etc.
- **content access modes:** content may be read-only or free for modification. Additional regulations can exist which constrain the access to content (authentication, rights, payment, ...).

In a content part multiple content objects may be given, e.g., for providing an image in three resolutions:

```
class EquestrianStatue {
  content highResImage: JPEGImageReference; //for print
          onlineImage:  JPEGImageReference; //for web site
          thumbnail:    JPEGImageContainer; //for preview
  concept ... }
```

The possibility of having more than one content container may also be used to have the content presented by different media, or to provide various images of the same content, e.g., photographs from different angles, with changing illumination, etc.

### 4.2 Conceptual Content Modeling: Asset Characteristics and Relationships

As already outlined in section 3 the concept part of an asset carries various contributions to cover the Peirce program for entity definition. On the level of firstness categories we provide the notion of characteristics. Characteristic attributes can be initialized by default values or objects (of basic or user-defined types). It is important to note, though, that characteristics are not intended for modeling object states but for describing the characteristics of entities.

```
class EquestrianStatue {
  content ...
  concept characteristic sex: {female, male};
          characteristic paintedAt: Date;
                          := new Date{1800 May 20th};
          characteristic title: String
                      := "Bonaparte crossing the Alps";
          characteristic photo: SmallJPEGImageCopy
                      := image.getThumbnail(...);
          ... }
```

Additionally, asset definitions allow the specification of constraints. They add to thirdness properties of assets. By giving a type, a domain restriction is already posed on the characteristics. It can be further narrowed by value or object constraints as in the following example:

```
...
characteristic paintedAt: Date
               < new Date{1800 September} and
               > new Date{1800 January};
characteristic title: String
               > "Bonaparte" or > "Napoleon";
characteristic photo: SmallJPEGImageCopy
               res < 100;
...
```

Here the characteristic attribute `paintedAt` is limited to the date interval `January` to `September 1800`, and `title`, once set, has to contain either the string literal "`Bonaparte`" or "`Napoleon`" (or both, "`Napoleon Bonaparte`"). Possible comparators test for equality ("="), lesser ("<"), greater (">"), different ("#"), or similar ("∼") values. How the comparator is evaluated depends on the object type. In the case of an object-valued characteristic it can be constrained in each of its attributes. In the above example `photo` may only have a resolution below 100.

Relationship attributes describe secondness properties of an entity. In the same way as for characteristics, default bindings and constraints can be given for relationships, but based on assets instead of values or objects:

```
class EquestrianStatue {
  content ...
  concept characteristic ...
          ...
          relationship paintedBy: Artist := AID2;
          //AID2 is asset for painter Jacques-Louis David
          relationship requestBy: Ruler := AID3;
          //AID3 is asset for emperor Napoleon Bonaparte
          relationship depicted:Person:=self.requestBy;
          //by default, the bindings to attributes depicted and
          //requestBy are made identical
          ... }
```

Technically relationships are references to other assets. Object-oriented programming languages hide the difference between attributes with value assignment and those with object bindings – i.e., firstness and secondness use of categories (languages like C$^{++}$ make a clear distinction but this is regarded as low-level programming since the choice for using a value or a pointer it is not motivated by the model).

In some conceptual modeling approaches a uniform treatment of attributes of kind "value assignment" and of kind "object binding" is considered an advantage. However, most object-oriented DDLs [9] as well as UML (via attributes and associations) and many other conceptual modeling languages make the kind of attribute explicit.

Our position in making an explicit distinction between attributes for characteristic values and objects and for relationships to assets is additionally motivated by the support a user gains from asset technology. While the characteristic attributes form the basis for value-based asset querying in the sense of database query languages, the relationship attributes support asset browsing and navigation.

## 4.3 Remarks on Asset Class Semantics

In our experiments with asset languages we distinguish two ways of defining asset semantics. As outlined so far our asset language has an *intensional* semantics, i.e., is given by (type) predicates which assets of a certain class have to fulfill. A second kind of asset class semantics could be called *extensional* because asset class semantics is based on prescribed asset instances.

**Intensional Asset Class Semantics.** Intensional asset class definitions are comparable to classes in object-oriented languages and are given by class declarations containing characteristic and relationship attributes with their constraints and default bindings.

As in object-oriented languages our Asset Definition Language allows class refinement. Refined subclasses inherit characteristics and relationships as well as content part from the underlying base class:

```
class EmployeeOfCompanyABC refines Employee {
  concept relationship employer: Company := ABC; }
```

Class definitions as discussed so far are considered static; they can be statically type-checked and compiled. If, however, a class is defined, for example, by the constraint, `Company = ABC`, instead of the initializing assignment, `Company := ABC`, dynamic type checking is required.

**Extensional Asset Class Semantics.** Extensional definitions of asset semantics is essentially given by collections of `prescribed` asset instances. Currently we are experimenting with collections which are either unordered (sets) or ordered (lists). A second criterion regulates the degree to which such prescribed collections definitively restrict asset instances or not. If a prescribed collection is marked as `final`, users can only select prescribed collection elements and the asset class definition is reduced to an enumeration type:

```
class EquestrianStatue {...}
  prescribed final {AID_1, ..., AID_n}
```

Prescribed collections may also be indicated as `initial` which means that users can start with any element of the collection and modify it within the boundaries drawn by the intensional part of the class definition.

In our flagship project on *Political Iconography* the asset class `Equestrian-Statue`, for example, is defined by

```
class EquestrianStatue {...}
  prescribed initial AID_1, ..., AID_n
```

where the list indicates that order is considered relevant (following Aby Warburg's principle of *good neighborhood*). By starting with a specific element from the list, say, $\texttt{AID}_i$, a user indicates that – subjectively – this element is considered "closest" to the new asset instance he wants to create.

Examples of host systems exploiting asset class definitions based on prescribed collections are automatic content classifiers for which such instance collections serve as training sets [3, 13, 52], or e-learning environments presenting the example sets to students (see section 7).

## 4.4 On Signification Services

A particular service expected from an asset instance are contributions to the identification of the entity which it represents. In our example the value of a distinctive characteristic asset attribute, say, the attribute `photo` which is computed from the asset's content, may serve as an *iconic* entity signifier. Other characteristics may play the role of *indexical* signifiers. The network of related assets – classes and instances – supports the notion of *symbolic* signifiers [16].

**Iconic Signification.** Iconic signification is achieved through firstness categories [29]. An iconic signifier resembles for a user some similarity with some entity. An iconic signifier *re-presents* the associated entity and brings it into the user's mind. Those sets of characteristics which make the asset an icon of some entity are distinguished by the keyword `icon`. In the example below the characteristic attribute `photo` of an instance of asset class `EquestrianStatue` signifies some entity iconically. The characteristic `sex`, however, does not provide any signification service.

**Indexical Signification.** Indexical signification is, one way or the other, based on the notion of *co-occurrence* and is closely related to Peirce secondness categories [29, 16]. There needs to be some matchmaking circumstance by which an indexical signifier and its signified entity are brought together. Asset instances provide, by definition, such co-occurrence between contents and concepts, and asset relationship attributes establish co-occurrence when bound to other asset instances. Furthermore, assets are supposed to support indexical signification of application entities by asset characteristics as, for example, `registrationNo`:

```
class EquestrianStatue {
  content ...
  concept characteristic sex: {female, male}; ...
          characteristic photo: SmallJPEGImageCopy;
          characteristic registrationNo: Integer;
          ...
          icon  photo;
          index registrationNo; }
```

**Symbolic Signification.** Symbolic signification makes full use of the categorical structures introduces by asset classes and type systems and is, therefore, closely related to Peirce's thirdness level.

Classes, types and constraints contribute to symbolic signification. If, for example, the definition of `EquestrianStatue` includes a relationship to artists which are constrained to the epoch "Renaissance", then renaissance statues may be signified symbolically via artists.

Well-structured interfaces to large-scale content management systems make extensive use of symbolic signification [17, 48].

## 5 On Asset System Responsiveness

As equally important as the expressiveness of an asset language for entity modeling are the *modalities* by which such expressive models can be utilized, i.e., their *responsiveness.* According to Cassirer [8, 47] and others (see, for example, [35]), responsive entity modeling must be

- *open*, i.e., the categories used for entity modeling must not be pre-defined by fixed ontologies but need to be open for adaptation by specialization or generalization as demanded by the application entities at hand;
- *dynamic*, i.e., any aspect of an entity model – all related asset instances as well as their class and type definitions etc. – must be accessible, evaluable and adaptable at any time.

Openness and dynamics together allow asset systems to be constantly adapted, refined and personalized in a process which converges towards the requirements as demanded by its users' tasks.

### 5.1 Asset System Openness

In the 1920ies Ernst Cassirer already strongly requested that content-concept pairs – which he calls *symbols* [8, 47] – be formed in an open context with no restriction to predefined, fixed ontologies for concepts and categories. Humans – Cassirer's *animal symbolicus* – are able to grasp and communicate entities and their media and model views adequately only if concepts can be openly specialized and generalized depending on the modeling needs of the entity at hand. In fact Cassirer considers such differentiation efforts as *the* essential contribution to entity modeling.

In programming, conceptual openness was neglected until the late 60ies when the programming language Simula [14] first introduced object-oriented principles into software simulation systems.

Our notion of assets corresponds closely to the notion of object. While "the *object-oriented* approach to programming is based on an intuitive correspondence between a software simulation of a physical system and the physical system itself" [1], our *asset-oriented* approach to entity modeling aims at an intuitive correspondence between a software representation of a perceptible domain and the perceptible domain itself.

For the domain of *Political Iconography*, for example, our asset application system WEL ([34], see section 7) represents thousands of political concepts related to hundreds of thousands of asset instances which serve essentially as iconic representations of such political concepts.

[1] claims that for applications such as physical systems simulation "objects form natural data abstractions boundaries and help focus on system structure instead of algorithms". For asset-orientation we make a similar claim by referring to the substantially improved

- analogy between asset models and application domains;
- resilience of the asset models;
- reusability of the components of the asset model.

The major single property which supports such demands is the reusability of asset components, i.e., the easy use of an asset in more than one context. As for objects we demand, for example, that asset classes be reused by importing them into other classes and a generic asset class be reused by instantiating it with different parameters.

The subsequent asset class definition specializes equestrian statues depicting only female equestrians:

```
class FemaleEquestrianStatue refines EquestrianStatue{
  concept characteristic sex: {f,m} = f; ... }
```

The technology (see section 6) required for asset system openness resembles to some extent modern object-oriented language compiler technology. However, it also has to address the issue of co-existing populated asset schemata and their cooperation.

### 5.2 Asset System Dynamics

Besides openness there is a second demand, also already strongly requested by Cassirer. He argues for a maximum of dynamic support for the process of asset system use and improvement. This request is in contrast to systems which force users to leave their actual working process and go through lengthy phases of redefinition, redesign, conversion, etc. Only an open *and* dynamic asset system can be adapted, refined, recompiled, personalized etc., towards the requirements needed by its users' tasks.

A prerequisite of a dynamic asset system is online asset evaluation. While the well-structured concept aspect of assets – asset instance as well as class definitions – can benefit from database query technology, the semi-structured content aspect of assets requires search and indexing technology from information retrieval. Asset systems offer interfaces for combined queries running against both sides, asset content and concepts.

## 6 Asset Compilation and Configuration Technology

Following our extensive tradition in persistent languages R&D [30, 33, 24] we are intensely involved in the development of software systems for asset language environments [31]. Our demand for asset system responsiveness in the above sense requires dynamic openness as well as support for asset re-usability and sharing. Such demands disallow asset systems implementation by conventional compiler component and phasing technology:

– dynamic openness implies the need for runtime redefinition and inspection of assets and of their implementing components each time a user supplies a new asset definition or personalizes an existing one;
– re-usability and sharing requires that two asset system components have to cooperate if somebody provides assets for a domain which somebody else wants to use. Such cooperation structures change dynamically with asset definitions;
– working with personalized asset definitions involves that their redefined components have to be able to deal with existing asset instances created according to a schema used previously.

For asset systems implementation we take a two-step approach: the first step is driven by an asset model compiler, the second is based on a module configurator. The compiler translates ADL definitions into a set of modules of different kinds which form the basis for implementing that model. The configurator creates the modular structure of the executable target system for asset modeling and management. This way, we achieve our goal of an open dynamic asset system without paying the performance penalty for runtime interpretation [15] which – as experience from other projects show – could be prohibitively high.

### 6.1 An Asset Language Compiler

Our model compiler translates ADL asset definitions into the object model of some programming language. The current prototype uses Java as its target language. The compiler generates a set of interfaces and classes which reflect the asset definitions through method declarations which the compiler invents based on the characteristics and the relationships of the assets.

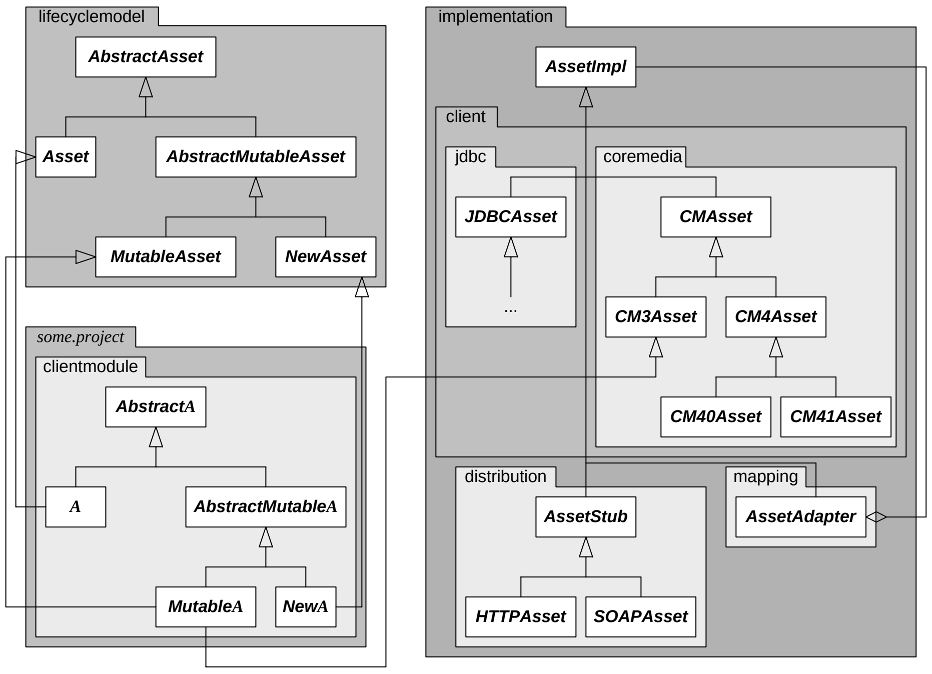

**Fig. 3.** Compiler generated objects

By and large, the method signatures conform to the JavaBeans standard [41], defining methods according to the name pattern `getC()` and `setC()` for a characteristic attribute $c$ and `addR()`, `removeR()`, and `hasR()` for a relationship attribute $r$. In fact several interfaces are created for each asset class which reflect certain life cycle states of the assets, each with life cycle methods to change state (for a thorough overview see fig. 3).

Besides the interfaces for the assets themselves a set of auxiliary interfaces is generated:

– iterators [18] to handle collections of assets,
– factories for creating asset instances according to the design pattern “factory method” [18],
– query interfaces for creating query objects, equipping them with the query expression and evaluating them to iterators, and finally
– visitors [18] to distinguish between the possible classes (instead of having a “type switch” operator [1]) of an asset and to determine an asset’s state in a type-safe manner.

The interfaces are implemented by classes which reflect the kind of a module (see section 6.2), thus separating relevant concerns of asset system construction. Asset constraints are compiled into the modification methods (set, add, remove), thus having them checked each time a modification is requested (in terms of JavaBeans: they are constrained properties; see also [23]). In addition the modification methods send notifications whenever a modification is applied (bound properties). The value and binding initializations are compiled into the factory methods. In principle all methods of all interfaces are implemented in a way that they can be configured for auditing each invocation to support monitoring of work in progress and archival of works.

As a result of the open dynamics of asset systems the model compiler accepts asset declarations in two modes. One is to simply compile ADL statements as described throughout this paper. In this case the compiler generates the interfaces and classes necessary for the modules of a desired system. To make use of the openness the compiler can also be told that an asset definition refines an existing model. In this case the compiler first computes the integrated model derived from the existing and the new asset declarations and then builds the interfaces and classes. In addition, further classes for mapping between the models are generated for the mapping component outlined in section 6.2.

The compiler has a single front-end for parsing and checking asset definitions in the ADL. For the various kinds of modules to be produced there is one back-end for each one. To be able to generate asset systems with different back-end configurations the compiler is appropriately parameterized. In ongoing research we are working on growing sets of such back-end configurations aiming at a pattern library for asset system generation. For additional technical details please refer to [36].

## 6.2 On Asset System Modules

For asset system implementation we distinguish several *kinds* of modules. The model compiler creates sets of classes for each instance of a module kind. They form the basis for a domain-specific software architecture with a generated implementation [50]. Since the software artifacts are generated to work in concert we call them *modules*, however, in fact they enjoy essential properties of software *components* [43, 2], the most important one being their statelessness.

Statelessness is an essential prerequisite for module exchange at runtime. Statelessness is relevant because the demand for dynamic openness leads to runtime module modification whenever a user changes its asset model. In such cases the model compiler is employed, and the modules built are used to reconfigure the asset management system.

As mentioned above the model compiler adds individual methods to the object classes generated for asset classes. To allow random module combinations all modules have a uniform interface. It is generic so that applications developed against the module interface do not break due to changes requested by dynamic openness. The specific requirements of a concrete model are reflected by the asset parameters of the modules' operations.

So far we have identified five kinds of modules for asset system implementation (sketched in fig. 4): client modules and server modules as well as modules for mediation, distribution and mapping. These alternatives have satisfied all demands of our current conceptual content management projects, i.e., demands

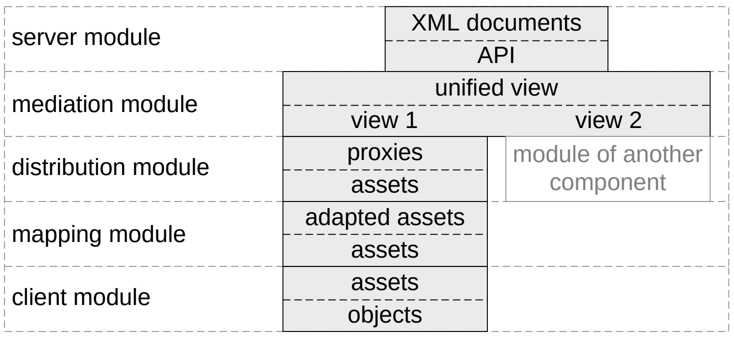

**Fig. 4.** Asset system module configuration

from dynamic openness as well as from non-monotonic and monotonic personalization, replication, access control, etc.

**Modules for Asset Clients and for Asset Servers.** Modules of the first kind – client modules – are based on persistence and retrieval services as provided by standard components. The individual contribution of a client module is the mapping of objects which represent assets down to the employed standard component and, conversely, the mapping of data retrieved from that component up into the asset model.

ADL expressions serve as abstract descriptions of both application and the data layer (comparable, for example, to the interface modules of [22]). There may be different compiler back-ends for different off-the-shelf technologies, e.g., different database systems. Currently we use JDBC [42] and file access (the top of the generator hierarchy for JDBC can be seen in fig. 3, package `jdbc`). The back-ends contain the knowledge on how to create a mapping for a particular software product.

The classes generated from an ADL expression to form a client module have to be able to cope with the fact that schemas evolve and that there is data created for outdated schema versions [4]. Since the third-party systems we are using are not capable of allowing populated and versioned schemas to coexist and having the same functions applied to both of them, evolution has to be handled by the asset management system [27]. Old client modules (and the standard component they use) are conserved while fresh client modules for the new model are generated. To access multiple base systems in a uniform way, a mediation module (see next section) is employed. Client and mediation modules together form mediators in the sense of [51].

Server modules are the counterpart of client modules and offer the asset management system functionality as a service to clients. For clients to be able to use such services a standard protocol has to be applied. So far we use XML documents with a schema generated to reflect the asset definitions and transport them by means of HTTP. We have begun work on server module development for WebServices [5] with generated descriptions given in the WDL [11].

**Modules for Mediation, Distribution and Mapping.** The are three further kinds of modules which are mentioned only briefly in this paper: mediation, distribution and mapping modules.

Mediation modules are capable of delegating calls to other modules and of combining their responses in different ways. They are the crucial part of many module configurations, especially those involving openness.

Distribution modules offer remote communication between two components. They consist of two parts, one for each of the communication partners. Their service is similar to RPC and the two parts correspond to stubs and skeletons. However, instead of using a standardized marshalling format they exchange XML documents with a generated schema (comparable to the suggestion of [37]).

A final module kind which is highly relevant to asset system openness are mapping modules. They support model mapping [4] thus allowing modules generated from different ADL schema revisions to communicate. A mapping module is made up of adapters [18] which convert assets according to the ADL models involved. Adapters wrap an object of a class generated for a base model and fulfill the interface required by the derived model (see package `mapping` in fig. 3).

Mapping issues are covered by a module kind of its own rather than integrating it into the other kinds of modules so that it can be plugged dynamically [26].

## 7 Perspective: Interdisciplinary Projects in Conceptual Content Management

The field of algorithmic programming has had from the very beginning a rather clear understanding of its underlying computational models. Nevertheless, experience from countless software engineering projects have been necessary to provide the basis for the development of modern high-level programming languages, their efficient implementation technology and their effective environments for large-scale software development. Following similar arguments we see an urgent need for extensive interdisciplinary project-based experience in conceptual content modeling and management.

Many of the insights leading to the asset approach presented here have been gained in the project *Warburg Electronic Library* [34, 7]. The WEL system [48] was developed in an interdisciplinary joint project between the Art History department at Hamburg University and our Software Systems Institute. By now the WEL system has matured to a productive system that is used by hundreds of researchers internationally and is being extended in cooperation with several institutions.

The interdisciplinary WEL project was founded right from the start on extensive domain material on *Political Iconography* and on immense user experience from the area of art history. In the meantime, further content collections and methodological experiences have been provided by other project partners, e.g., from the area of commercial advertisements or from media industry.

Political Iconography (PI) proved to be a perfect application domain for an interdisciplinary project on conceptual content modeling and management. Basically, PI seeks to capture the semantics of key concepts in the political realm under the assumption that political goals, roles, values, means, etc. require mass communication which is implemented by the iconographic use of image-oriented content.

Martin Warnke, our project partner in art history, started his (paper-based) work on PI in the early 80ies. To date he and his colleagues have identified about 1,500 named political concepts and collected more than 300,000 records on iconographic works relevant to PI. In 1990 Warnke was recognized for his work by the Leibniz-Preis, one of the most prestigious research grants in Germany.

A major use of the WEL is its application for educational purposes in e-learning scenarios [25] exploiting its advanced functionality for

– customization of assets to establish thematic views on a domain, and
– personalization of the thematic views by students to construct individualized views.

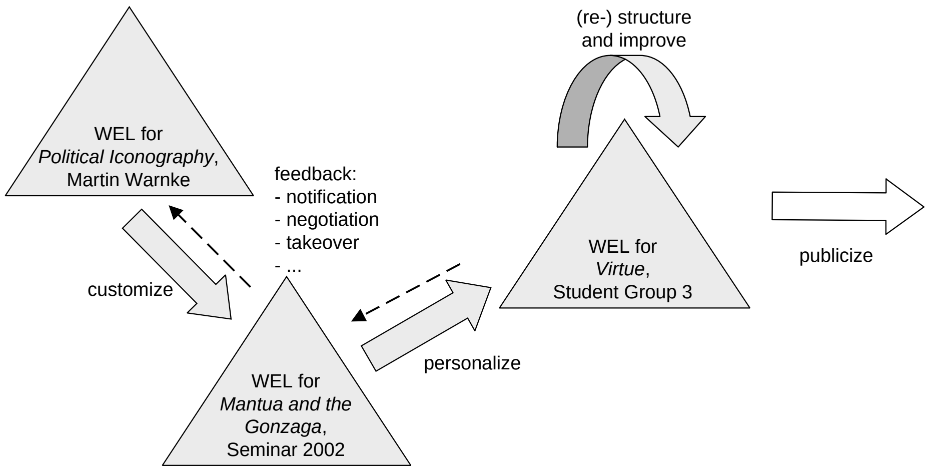

**Fig. 5.** Asset system for Art History education: A seminar on *Mantua and the Gonzaga*

Fig. 5 outlines the use of customization and personalization in an art history seminar on *Mantua and the Gonzaga*. The all-encompassing PI asset collection (owned by art historian Warnke) is first customized into an asset collection for the seminar project *Mantua and the Gonzaga* (owned by the supervising research assistants). The main objective of individual student projects in that seminar is to further customize and extend the seminar assets, structurally as well as content-wise. Publicizing the final content in some form of media document – a traditional report or an interactive web page – constitutes another educational objective of the seminar.

We conclude by pointing out that many web application projects are essentially online content management projects. Therefore, most of our R&D projects [17, 28, 44] as well as some of our commercial activities [12, 20] profit substantially from our insight in conceptual content modeling and management.

**Acknowledgement.** We would like to express our thanks to the members of the Software Systems Institute, in particular, Rainer Müller, Michael Skusa, Ulrike Steffens and Axel Wienberg, and Florian Matthes, now Technical University of Munich, Germany. Last but not least our warm thanks to Professor Martin Warnke, Art History Department at the University of Hamburg.

## References

1. Martín Abadi and Luca Cardelli. *A Theory of Objects.* Monographs in Computer Science. Springer-Verlag New York, Inc., 1996.
2. U. Aßmann. *Invasive Software Composition.* Springer-Verlag, 2003.
3. Stefan Berchthold, Bernhard Ertl, Daniel A. Keim, Hans-Peter Kriegel, and Thomas Seidl. Fast Nearest Neighbor Search in High-Dimensional Spaces. In *Proc. 14th IEEE Conf. Data Engineering.* IEEE Computer Society, 1998.
4. Philip A. Bernstein and Erhard Rahm. Data Warehouse Scenarios for Model Management. In Alberto H. F. Laender, Stephen W. Liddle, and Veda C. Storey, editors, *Proc. 19th International Conference on Conceptual Modeling*, volume 1920 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2000.
5. David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. Web Services Architecture, W3C Working Draft. http://www.w3.org/TR/2003/WD-ws-arch-20030808/.
6. Michael L. Brodie, John Mylopoulos, and Joachim W. Schmidt, editors. *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages.* Topics in Information Systems. Springer-Verlag, 1984.
7. Matthias Bruhn. The Warburg Electronic Library in Hamburg: A Digital Index of Political Iconography. *Visual Resources*, XV:405–423, 2000.
8. Ernst Cassirer. *Die Sprache, Das mythische Denken, Phänomenologie der Erkenntnis*, volume 11-13 Philosophie der symbolischen Formen of *Gesammelte Werke.* Felix Meiner Verlag GmbH, Hamburger Ausgabe edition, 2001-2002.
9. R.G.G. Cattel, Douglas Barry, Mark Berler, Jeff Eastman, David Jordan, Craig Russell, Olaf Schadow, Torsten Stanienda, and Fernando Velez, editors. *The Object Database Standard: ODMG 3.0.* Morgan Kaufmann, 2000.
10. Peter P. Chen. The Enity-Relationship Model: Toward a Unified View of Data. In Douglas S. Kerr, editor, *Proceedings of the International Conference on Very Large Data Bases, September 22-24, 1975, Framingham, Massachusetts, USA*, page 173. ACM, 1975.
11. Roberto Chinnici, Martin Gudgin, Jean-Jacques Moreau, and Sanjiva Weerawarana. Web Services Description Language (WSDL) Version 1.2 Part 1: Core Language. www.w3.org/TR/wsdl12/, June 2003.

12. Homepage of the CoreMedia© AG. www.coremedia.com, 2003.
13. Nello Cristianini and John Shawe-Taylor. *An Introduction to Support Vector Machines (and other kernel-based learning methods).* Cambridge University Press, 2000.
14. O. Dahl and K. Nygaard. Simula, an Algol-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.
15. C.J. Date. *What Not How – The Business Rules Approach to Application Development.* Addison-Wesley, 2000.
16. Terrence W. Deacon. *The Symbolic Species: The Co-evolution of Language and the Brain.* W. W. Norton & Company, Inc., 1997.
17. EURIFT Information Portal. www.eurift.net, 2003.
18. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.
19. Eric C.R. Hehner. *A Practical Theory of Programming.* Monographs in Computer Science. Springer-Verlag, 1993.
20. Homepage of the infoAsset© AG. www.infoasset.de, 2003.
21. I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process.* Addison-Wesley, 1999.
22. Manfred A. Jeusfeld. Generating Queries from Complex Type Definitions. In Franz Baader, Martin Buchheit, Manfred A. Jeusfeld, and Werner Nutt, editors, *Reasoning about Structured Objects: Knowledge Representation Meets Databases, Proceedings of 1st Workshop KRDB'94,*, volume 1 of *CEUR Workshop Proceedings*, 1994.
23. H. Knublauch, M. Sedlmayr, and T. Rose. Design Patterns for the Implementation of Constraints on JavaBeans. In *NetObjectDays2000, Erfurt, Germany.* 2000.
24. F. Matthes, G. Schröder, and J.W. Schmidt. Tycoon: A Scalable and Interoperable Persistent System Environment. In Malcom P. Atkinson and Ray Welland, editors, *Fully Integrated Data Environments*, ESPRIT Basic Research Series, pages 365–381. Springer-Verlag, 2000.
25. Hermann Maurer and Jennifer Lennon. Digital Libraries as Learning and Teaching Support. *Journal of Universal Computer Science*, 1(11):719–727, 1995.
26. Mira Mezini, Linda Seiter, and Karl Lieberherr. Component integration with pluggable composite adapters. In *Software Architectures and Component Technology.* Kluwer, 2000.
27. Giorgio De Michelis, Eric Dubois, Matthias Jarke, Florian Matthes, John Mylopoulos, Joachim W. Schmidt, Carson Woo, and Eric Yu. A Three-Faceted View of Information Systems. *Communications of the ACM*, 41(12):64–70, 1998.
28. Rainer Müller, Claudia Niederée, and Joachim W. Schmidt. Design Principles for Internet Community Information Gateways: MARINFO – A Case Study for a Maritime Information Infrastructure. In V. Bertram, editor, *Proceedings of the 1st International Conference on Computer Applications and Information Technology in the Maritime Industries (COMPIT 2000)*, pages 302–322, April 2000.
29. C.S. Peirce. *Collected Papers of Charles Sanders Peirce.* Harvard University Press, Cambridge, 1931.
30. Joachim W. Schmidt. Some High Level Language Constructs for Data of Type Relation. *ACM Transactions on Database Systems*, 2(3):247–261, 1977.
31. Joachim W. Schmidt, Gerald Schröder, Claudia Niederée, and Florian Matthes. Linguistic and Architectural Requirements for Personalized Digital Libraries. *International Journal on Digital Libraries*, 1(1):89–104, 1997.

32. Joachim W. Schmidt and Hans-Werner Sehring. Dockets: A Model for Adding Value to Content. In Jacky Akoka, Mokrane Bouzeghoub, Isabelle Comyn-Wattiau, and Elisabeth Métais, editors, *Proceedings of the 18th International Conference on Conceptual Modeling*, volume 1728 of *Lecture Notes in Computer Science*, pages 248–262. Springer-Verlag, November 1999.
33. J.W. Schmidt and F. Matthes. The DBPL Project: Advances in Modular Database Programming. *Information Systems*, 19(2):121–140, 1994.
34. J.W. Schmidt, H.-W. Sehring, M. Skusa, and A. Wienberg. Subject-Oriented Work: Lessons Learned from an Interdisciplinary Content Management Project. In Albertas Caplinskas and Johann Eder, editors, *Advances in Databases and Information Systems, 5th East European Conference, ADBIS 2001*, volume 2151 of *Lecture Notes in Computer Science*, pages 3–26. Springer, September 2001.
35. Christiane Schmitz-Rigal. *Die Kunst offenen Wissens, Ernst Cassirers Epistemologie und Deutung der modernen Physik*, volume 7 of *Cassirer-Forschungen*. Ernst Meiner Verlag, Hamburg, 2002.
36. Hans-Werner Sehring. *Konzeptorientiertes Content Management: Modell, Systemarchitektur und Prototypen*. PhD thesis, Arbeitsbereich Softwaresysteme, Technische Universität Hamburg-Harburg, Deutschland, 2003.
37. German Shegalov, Michael Gillmann, and Gerhard Weikum. XML-enabled workflow management for e-services across heterogeneous platforms. *The VLDB Journal*, 10(1):91–103, 2001.
38. John Miles Smith and Diane C. P. Smith. Database abstractions: Aggregation. *Communications of the ACM*, 20(6):405–413, 1977.
39. John F. Sowa. *Knowledge Representation, Logical, Philosophical, and Computational Foundations*. Brooks/Cole, Thomson Learning, 2000.
40. W. Stallings. *Networking Standards: A Guide to OSI, ISDN, LAN, and MAN Standards*. Addison-Wesley, 1993.
41. Sun Microsystems. JavaBeans Specification. java.sun.com/products/javabeans/, 2003.
42. Sun Microsystems. JDBC Technology. java.sun.com/products/jdbc/, 2003.
43. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.
44. Hompage of the TeFIS project. www.sts.tu-harburg.de/projects/TuTechFoBe/, 1999.
45. Bernhard Thalheim. *Entity-Relationship Modeling: Foundations of Database Technology*. Springer-Verlag, 2000.
46. Unified Modeling Language Resource Center. www.rational.com/uml/, 2003.
47. Donald Verene, editor. *Ernst Cassirer: Symbol, Myth, and Culture. Essays and Lectures of Ernst Cassirer 1935-1945*. Yale University Press, 1979.
48. Homepage of the Warburg Electronic Library. www.welib.de, 2003.
49. Homepage of the Workflow Management Coalition. www.wfmc.com, 2003.
50. S. White and C. Lemus. Architecture Reuse Through a Domain Specific Language Generator. In *Proceedings of the Eighth Workshop on Institutionalizing Software Reuse*, 1997.
51. G. Wiederhold. Mediators in the Architecture of Future Information Systems. *IEEE Computer*, 25:38–49, 1992.
52. Yiming Yang and Jan O. Pedersen. A comparative study on feature selection in text categorization. In *Proc. 14th International Conference on Machine Learning*, pages 412–420. Morgan Kaufmann, 1997.