

Beyond Databases: An Asset Language for Conceptual Content Management

Hans-Werner Sehring and Joachim W. Schmidt

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

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

Abstract. Innovative information systems such as content management systems and information brokers are designed to organize a complex mixture of *media content* – texts, images, maps, videos, ... – and to present it through domain specific *conceptual models*, for example, on sports, stock exchange, or art history.

In this paper we extend the currently dominating computational container models into a coherent content-concept model intended to capture more of the meaning – thereby improving the value – of content. Integrated content-concept views on entities are modeled using the notion of *assets*, and the rationale of our asset language is based on arguments for *open language expressiveness* [19] and *dynamic system responsiveness* [8]. In addition, we discuss our experiences with a component-based implementation technology which substantially simplifies the implementation of open and dynamic asset management systems.

1 Introduction: On Content-Concept Integration

Important classes of innovative information systems such as content management systems and information brokers are designed to organize a complex mixture of *media content* – texts, images, maps, videos, ... – and to present it through domain specific *conceptual models* [6], for example, on sports, stock exchange, or art history.

Traditional implementations of such information systems reflect this complexity through their software intricacy resulting from a heterogeneous mix of conventional database technology, various augmentations by text, image or geo functionality (or just by blobs) and through additional organizational principles from domain ontologies [25, 5].

In this paper we argue for a homogeneous basis for conceptual content management and present

- an integrated content-concept model – based on so-called *assets* [21] –,
- an asset language and its conceptual foundation [19], [8], as well as
- an implementation technology and architecture.

Our work is inventive essentially due to the following three contributions:

1. Content is *always* associated with its concept and represented by assets, i.e., *content-concept-pairs*. In a sense, assets generalize the notion of typed values or schema-constrained databases. Assets represent application entities, concrete or abstract ones;
2. Asset schemata are *open* in the sense that users can change asset attributes on-the-fly and any time, thus guaranteeing best correspondence with the entity-at-hand;
3. Asset management systems are *dynamic*, i.e., the system implementation changes dynamically following any on-the-fly modification of an asset schema; this requirement demands a specific system modularization and an innovative system architecture.

Our paper is structured as follows: after a short introduction of our asset language (section 2) we discuss the essentials of open and dynamic asset-based modeling and present an extensive example from the domain of art history (section 3). In section 4 we argue the benefits of asset compilation and its advantages for software system construction. The overall modularization and architecture of asset-based information systems are presented in section 5. We conclude with a short summary and an outlook into further applications of asset-based technology.

2 An Asset Language for Integrated Information Management

Assets represent application entities by content-concept-pairs (fig. 1). Following observations of [8] and others, neither content nor a conceptual model of an entity can exist in isolation. The conceptual part is needed to explain the way content refers to an entity. Content serves as an existential proof of the validity of concepts.

The content facet of assets is managed through object-oriented multimedia container technology. Assets contain handle objects referring to pieces of content.

To support *expressive* entity representations the concept facet is given by three contributions:

1. characteristic values,
2. relationships between assets, and
3. rules (types, constraints, ...).

Characteristic values describe entities by their immanent properties. Though the values may change one value is always assigned. Relationships between assets describe entities by their relation to others. Such relations may be changed or – in contrast to characteristic values – even be removed. Regular assertions describe facts about a set of similar assets. Type and value constraints on characteristics and relationships fall in this category.

Thus, the notion of assets follows closely the theoretical work of [19] (firstness, secondness, thirdness) and [8] (indivisibility of content and concept).

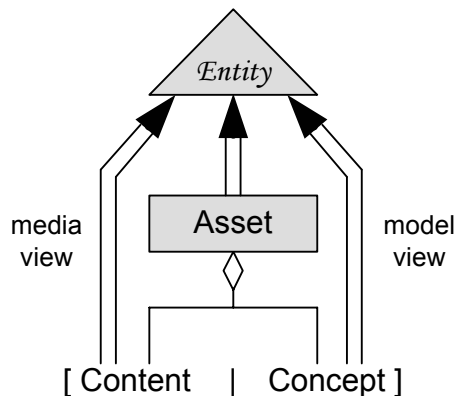


Fig. 1. Assets represent entities by [content | concept]-pairs

As already argued in the introduction, a conceptual content management system has to be based on a *responsive* dynamic model to adequately represent changing entities [21]. For expressive entity description and responsive domain modeling we propose an asset language to be employed to notate individual asset definitions and their systematics. Our asset language syntax corresponds to modern class-based languages [1].

An alternative to such a linguistic approach would be the use of a generic model to which the intended domain model is mapped. In this case users would have to face all the problems implied by an on-the-fly translation between their intended entity model and the generic model (“mentally compile and decipher” [10]).

As an *asset definition language* the language is used to define asset classes. The following code gives an example of an asset class definition:

```

class Equestrian {
  content reproduction : java.awt.Image ...
  concept
    characteristic yearOfCreation : java.util.Calendar
    characteristic medium : de.tuhh.sts.wel.Media
    relationship painter : Artist
    relationship epoch : Epoch
    constraint epoch = painter.epoch
    ...
}
  
```

The body of a class definition contains two sections. Under *content* references to pieces of content are defined by content handles. Their type is given in the class definition. Valid types are defined by some object-oriented language underlying the asset language. Currently we use Java for this purpose.

The conceptual part of entity descriptions is formalized in the `concept` section. Here the contributions discussed above can be found: characteristics, relationships, and rules.

Definitions starting with `characteristic` and `relationship` define attributes which can be set for any instance of a defined class. Each of these is identified by a name. A type for actual values or bindings is given after the colon. In the case of characteristics this is a Java class again. In the example shown above the standard Java class `Calendar` and the application-specific class `Media` are used. For relationships an asset class is given which constrains the type of assets referred to. If the class name is followed by an asterisk (“*”) it is a many-to-many relationship binding a set of asset instances.

Constraints pose value restrictions on the attributes of all instances of one asset class. In the above example it is defined that the epoch of the equestrian artwork has to be the same as the epoch of the artist (if one is bound). In constraint statements all attributes of the current class can be used plus the attributes of bound assets.

In the example the epoch binding for the current `Equestrian` instance is compared to the corresponding binding of the related `Artist` instance. For this to work the asset class `Artist` needs to define an epoch relationship to `Epoch` assets, just like `Equestrian`.

Possible comparators in constraint expressions test for equality (“=”), lesser (“<”), greater (“>”), different (“#”), or similar (“~”) values or bindings. How the comparator is actually evaluated depends on the compared attributes’ types. For characteristics, evaluation is done according to a Java `Comparator`. For relationships the comparisons are mapped to set relations (equality, inclusion, ...). In both cases, similarity is decided in an implementation-dependent manner (see below). Expressions can be combined using the logical operators `and`, `or`, and `not`.

Classes can be defined as subclasses of existing ones using the `refines` keyword:

```
class MedievalEquestrian refines Equestrian {  
    concept constraint epoch = middleAges  
}
```

This way definitions are inherited by the subclass. Here, a constraint on epoch is added. Inherited definitions can be overridden.

Another way to define asset classes is by giving an extensional definition. This is done by naming a set of asset instances which define a class. There are two variants of the extensional class definition. The first one gives a fixed set of instances which are the complete extent of an asset class:

```
class Equestrian definedby { e1, e2, e3 }
```

This way the asset class `Equestrian` is introduced as an enumeration type with possible instances `e1`, `e2`, and `e3`.

For the second variant the set of asset instances serves as an example for the intended extent of the new asset class. A definition like

```
class Equestrian definedby ~ { e1, e2, e3 }
```

defines `Equestrian` to be the class of all instances similar to `e1`, `e2`, and `e3`. To decide upon similarity, conceptual content management systems incorporate retrieval technology [3]. The set of asset instances is used as a training set.

For the management of asset instances statements of the asset language serve as an *asset query and manipulation language*. A `create` operation is used for the creation of asset instances. The following is an example for the instantiation of an `Equestrian` instance:

```
create Equestrian {  
    medium := de.tuhh.sts.wel.Media.STATUE  
    painter := rubens  
}
```

Here, `STATUE` is a constant class variable of `Media` holding a Java object for the media type “Statue” (singleton [12]). `rubens` is the name of an `Artist` asset instance.

A variant of the `create` operation allows to name a prototype instance instead of the set of initial bindings: `create Equestrian eqProto`

For updates the `modify` operation is used. For example, the update of an `Equestrian` instance named `eq1` is done by:

```
modify eq1 {  
    medium := de.tuhh.sts.wel.Media.STATUE  
    painter := rubens  
}
```

A variant similar to that of `create` allows the naming of a prototype instance instead of the set of new value and instance bindings: `modify eq1 eqProto`

The `lookfor` operation is used to retrieve asset instances. It searches for all instances of a given class. As query parameters all characteristic and relationship attributes can be constrained. An example query for `Equestrian` instances which are statues by Rubens is:

```
lookfor Equestrian {  
    medium = de.tuhh.sts.wel.Media.STATUE  
    painter = rubens  
}
```

Due to space limitations not all aspects of the asset language can be explained in this section. The detailed definition of the asset language can be found in [23].

3 Asset-based Modeling: A Case for Open and Dynamic Information Systems

Asset definitions usually depend on considerations like the state of the entities to describe, the users’ expertise, their current task, etc. For various reasons such influencing factors may change over time (see [8]):

- The observed entities change. Thus, their descriptions have to be adjusted. This is true even for class definitions because in different states an entity is described by different sets of contents, characteristics, relationships, and constraints.
- The users’ expertise influences their information needs. Usually users are not willing to (explicitly) provide data they do not consider interesting. For communication with others, though, assets need to be tailored to the receiver’s needs.
- A user can view an entity while being in different contexts, e.g., depending on the task for which an asset is needed. Different asset definitions may be needed when changing context.

Thus, openness and dynamics as defined in the introduction are important properties of conceptual content management systems for a variety of reasons.

In application projects we observed that knowledge about application entities is captured by modeling the processes in which they have been created and used. Soft-goals like reasons, intentions, etc. of the creation of entities are recorded in such applications (see also [29]).

In the project *Warburg Electronic Library (WEL)* a prototypical open dynamic conceptual content management system has been developed [22]. In application projects our project partners create large numbers of assets modeling their domain. One primary application is art history [7]. Our project partners from art history use the WEL to pursue research in the field of Political Iconography. For content they collect reproductions of artworks (see fig. 2(a)).



(a) Media Content for the Concept “Equestrian Statue”

title	“Bonaparte Crossing the Alps...”
artist	Jacques-Louis David : Painter
regent	Napoleon I. : Emperor
motives	mountain, alps, horse, hand
text	Bonaparte, Hannibal, Carolus
reference	Carolus Magnus Crossing the Alps, Hannibal Crossing the Alps

(b) Conceptual Model of one “Equestrian Statue”

Fig. 2. Asset Facets

The conceptual part of assets records the historical events which prove that the artifact under consideration has been used to achieve political goals. Typical

information is the creation date and location of a piece of art, relationships to the regent ordering it and the artist who created it, relationships to other works which are influential, information on the way it has been presented, etc. (see fig. 2(b)). Sets of asset instances define categories which name political phenomena. Additional constraints reveal how products of art work for political reasons.

From a computer science perspective the WEL is an important project for understanding the nature of conceptual content management systems. It serves as a field study with students and scientists. The WEL has been online for several years now [27]. Its services are used by some hundred scientists worldwide, mainly from the humanities. In cooperation with our project partners it has been used as an e-learning tool in several seminars during the past years.

As a research tool the WEL maintains a set of assets available to a research community. Researchers employ openness and dynamics to model their hypotheses. They can do so without interfering with others and without allowing them to see their results. When there are valuable findings a community can choose to integrate the assets of one of its members. For this, the WEL maintains open asset models and the corresponding asset instances on a per-user basis. Within scopes controlled through group membership asset instances can be shared among users.

In e-learning scenarios assets are prepared as course material. The body of asset instances is maintained for teaching purposes or, as is the case with the WEL, research efforts being carried out. Openness is needed by both teachers and students [ML95]. Teaching staff can select and adapt assets as teaching material for a course to be supported. In seminars and lab classes students can modify this material. Such a process is illustrated in fig. 3. This way, students get hands-on experience with the definition of concepts, the validation of models, the creation of content, etc.

4 Information System Construction by Asset Compilation

Domain experts formulate asset models using the asset language introduced in section 2. As discussed in the previous section there is a demand for open systems allowing the modification of existing assets. For asset management systems to meet the openness requirement these are automatically created based on such domain models. As part of the asset technology this is done by an *asset compiler*.

The compilation process bears a resemblance to modern software-engineering approaches like Model Driven Architecture (MDA) [18]. The asset compiler creates a platform independent model from a domain model. The platform independent model is then translated into a running software system. The translation of domain models is described in this section. The following section concentrates on actual implementations.

As a first step in the compilation process a data model is created from asset definitions given in the asset language (comparable, for example, to the interface modules of [14]). The current asset compiler uses Java as its target language. The data model consists of Java interfaces. Additional parameterizations of standard

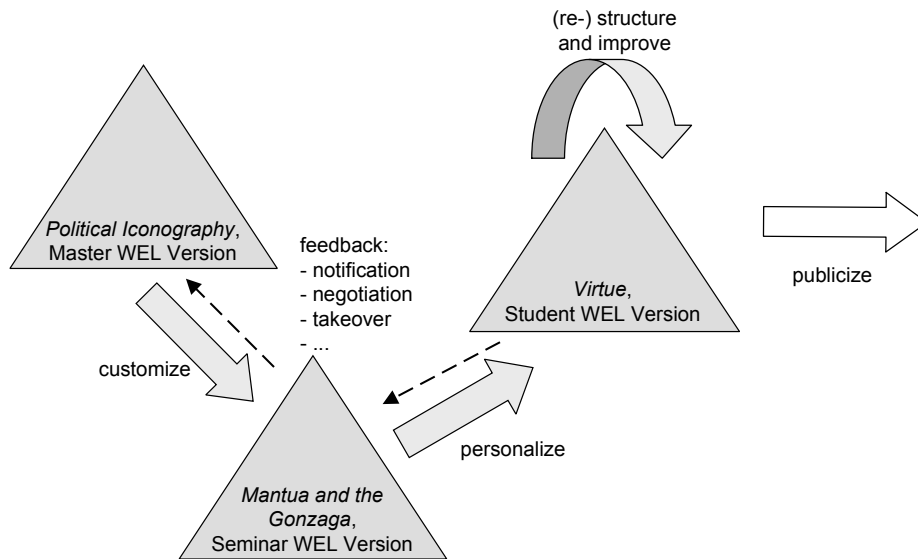


Fig. 3. Open Dynamic Asset Use in the WEL for E-Learning

components are created as needed (see next section). Examples are schemata for databases or content management systems, XML schemata, etc.

For each asset class a Java interface is created. Definitions of subclasses are mapped to subtypes. The interfaces adhere to the JavaBeans standard [13]. Access methods (“getter” and “setter”) are defined for characteristics and relationships. Class-level (thirdness) contributions are implemented in the operations of classes generated according to the interface definitions: constraints are mapped to constrained properties which throw a `VetoException` when the constraint is violated (see also [15]). Rules are expressed by bound properties which cause the invocation of further methods under the condition set by a constraint.

The UML class diagram in fig. 4 gives an overview of the generated code. The packages `lifecyclemodel` and `implementation` contain generic interfaces and classes which are part of the runtime environment of a conceptual content management system. A package like `some.project` is generated by the asset compiler.

The interfaces from package `lifecyclemodel` reflect possible states in the life of an asset instance. They define methods which allow state transitions as shown in the state chart in fig. 5.

Interfaces reflecting an asset model are created as subtypes of those generic ones. In fig. 4 the interfaces shown in package `some.module` are created for a defined asset class *A*. These introduce methods which reflect the asset classes’ characteristics, relationships, and constraints as explained above.

Not shown in fig. 4 are additional interfaces which describe the management of asset instances:

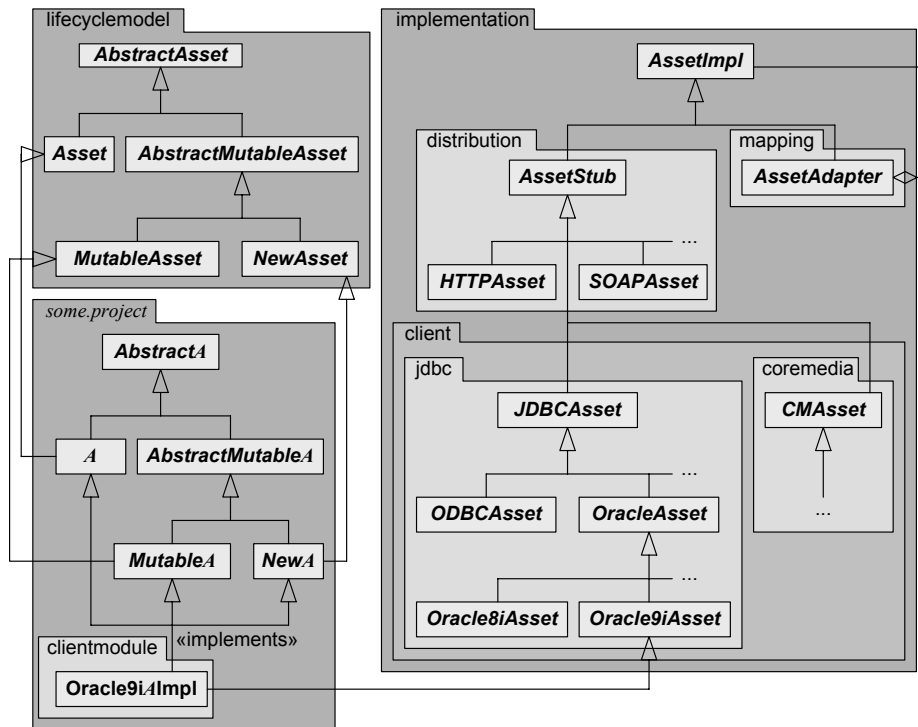


Fig. 4. Conceptual Class Diagram of Code Generated from Asset Definitions

- Class objects carry the asset class definitions into the data model (meta level). They offer reflection comparable to object-oriented programming languages.
- Instances are created following the factory method pattern [12].
- Query interfaces define possible queries to retrieve assets instances. They are equipped with methods to formulate query constraints. These constrain methods are generated for each defined characteristic and relationship and each comparison operator.
- For collections of asset instances iterators [12] are defined for each asset class.

The generated interfaces reflect the domain model. The abstract classes from the `implementation` package shown on the right of figure 4 introduce platform independent functionality. Classes which implement the interfaces and make use of the abstract classes are generated by the compiler. For an example see `AImpl` in package `clientmodule` in the class diagram. Sets of classes form modules which make up a conceptual content management system. These are described in the subsequent section.

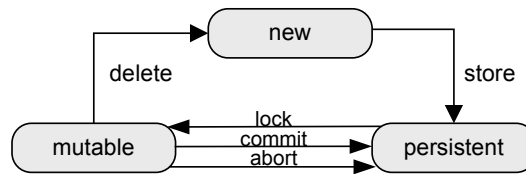


Fig. 5. State Diagram for the Asset Instance Life Cycle

5 Modular System Architectures for Asset Management

Open modeling allows users to adjust domain models at any time. This may affect the model of one user who wishes to change asset definitions, or the models of a user group and one of its users, who creates a personal variant of the group's model.

To dynamically adapt conceptual content management systems to changing models they are recompiled at runtime. The demand for dynamics leads to system evolution [17].

The evolution of conceptual content management systems has two aspects:

- the software needs to be modified, and
- existing asset instances need to be maintained.

Typical issues with respect to these two aspects of evolution are:

- Changes performed on behalf of individual users should not have any impact on others. Therefore, dynamic support for system evolution must not prevent continuous operation of the software system.
- On the one hand, assets as representations of domain entities cannot automatically be converted in general. On the other hand, manual instance conversion is not feasible for typical amounts of asset instances.
- If a user personalizes assets for his own needs, he still will be interested in changes applied to the original. Through awareness [11] measures he can be informed about such changes. To be able to review the changes, access to both the former and the current versions are needed. That is, revisions of assets and their schemata need to be maintained.

Crucial for both aspects of evolution – software as well as asset instances – is a modularized system architecture. On evolution steps distinguished system modules maintain sets of asset instances created under different schemata. They are produced by the asset compiler and dynamically added or replaced.

For our asset technology we identified a small set of module kinds of a conceptual content management system. The conceptual content management system architecture supports the dynamic combination of instances of the various module kinds. These modules share some similarities with components [26, 2] (combinability, statelessness, ...), but in contrast to these they are generated for a concrete software system. Modules constitute the minimal compilation units of the generated software which the compiler can add or replace.

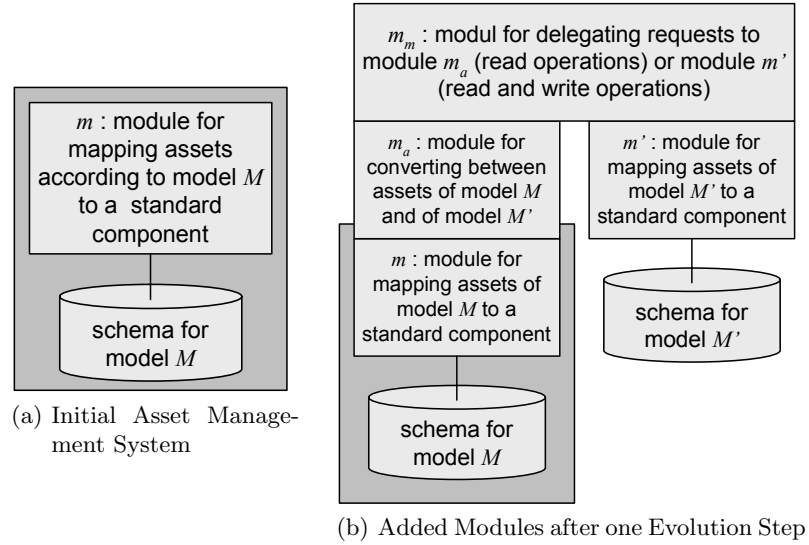


Fig. 6. Asset Management System Evolution Step

Figure 6 shows an example of the evolution of a user’s domain model. Assume that the conceptual content management system shown in fig. 6(a) is in operation. It simply consists of one module m as the application layer and a database as the data layer of a layered architecture. Both have been generated according to a domain model M .

If model M is redefined to become model M' the system is recompiled. This leads to the generation of additional modules which are incorporated for dynamic system evolution. The result is shown in fig. 6(b). First note that the original conceptual content management system is maintained as a subsystem of the new system version. This way existing asset instances are kept intact.

In this example a second database is set up to store asset instances following model M' . A module m' for accessing the database is created similar to m found in the original conceptual content management system. Two further modules are added to combine the two subsystems for models M and M' . These follow the mediator architecture [28], an important building block of the conceptual content management system architecture. The mapping module m_a serves as a wrapper lifting assets of M to M' (compare [20]). Mapping issues are covered by a module kind of its own rather than integrating it into the other kinds of modules so that mappings can be plugged dynamically [16]. The mediation module m_m reflects M' in the application layer of the new system version. It routes requests to either m_a or m' . Lookups are forwarded to both these modules and the results are unified. New asset instances are always created in m' according to M' . Update requests of instances of M lead to their deletion in the subsystem for M and their recreation in m' .

The two issues with evolution mentioned above are taken into consideration by this approach. Preserving the existing software as part of a new system version leaves it in continuous operation. With the mentioned update policy asset instances are incrementally converted when needed. This way, users can perform the task of reviewing the asset instances one by one. More sophisticated policies might take the mediation module to batch mode when only a certain amount of instances is left in the outdated schema.

In a similar way configurations for other functionality are set up. E.g., to store revisions of asset instances these are maintained by distinct subsystems. A mediation module takes care of the revision control.

The above example introduces the three most important module kinds: *client modules* to access standard components managing the assets' content and data, *mapping modules* to adjust schemata, and *mediation modules* to glue the modules of a conceptual content management system together. These form the core of any conceptual content management system. In addition to these, figure 7

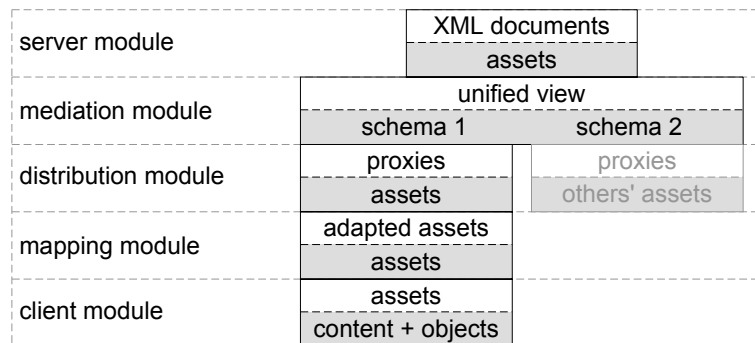


Fig. 7. Asset Management System: Module Kinds and Architectural Overview

shows the remaining two module types. *Distribution modules* allow the incorporation of modules residing on different networked computers. Fig. 4 indicates two possible implementations in package distribution: the HTTP-based transmission of XML documents with a schema generated from the asset definitions (comparable to the suggestion of [24]) and one for remote method calls using SOAP. *Server modules* (not shown in fig. 4) offer the services of a conceptual content management system following a standard protocol for use by third party systems. One example is a server module for Web Services [4] with generated descriptions given in the WDL [9].

6 Concluding Remarks

Our asset model abstracts and generalizes an essential part of the core experience in database design and information system development. Initial applications

demonstrate that asset-based modeling simplifies information system projects and increases the reusability of system functionality.

The degree of open schema and dynamic system changeability is substantially improved by a better understanding of the appropriate architecture and modularization of conceptual content management systems.

In addition we expect that asset-based modeling will greatly improve typical standard tasks in information systems administration. The very same methodology used for domain-specific entity modeling may also be applied to software entities and, therefore, to information systems themselves. Typical examples include naming and messaging services, user and rights management or visualization tasks. User interfaces, for example, will benefit significantly from an asset-based GUI model and UI description. A presentation logic which associates assets from the application domain and the GUI realm can then be used by a GUI engine to render such UI descriptions and exploit the dynamic openness of asset management for user interface adaptation.

References

1. Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag New York, Inc., 1996.
2. U. Almann. *Invasive Software Composition*. Springer-Verlag, 2003.
3. Thomas Büchner. Entwurf und Realisierung eines Java-Frameworks zur inhaltlichen Erschließung von Informationsobjekten. Master's thesis, Software Systems Department, Technical University Hamburg-Harburg, Germany, 2002.
4. David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. Web Services Architecture, W3C Working Group Note. <http://www.w3.org/TR/ws-arch/>, 11 February 2004.
5. Alex Borgida and Ronald J. Brachman. Conceptual Modeling with Description Logics. In Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider, editors, *The Description Logic Handbook: Theory, Implementation and Applications*, pages 349–372. Cambridge University Press, 2003.
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. Roberto Chinnici, Martin Gudgin, Jean-Jacques Moreau, Jeffrey Schlimmer, and Sanjiva Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. www.w3.org/TR/wsdl20/, March 2004.
10. Dov Dori. The Visual Semantic Web: Unifying Human and Machine Knowledge Representations with Object-Process Methodology. In Isabel F. Cruz, Vipul Kashyap, Stefan Decker, and Rainer Eckstein, editors, *Proceedings of SWDB'03, The first International Workshop on Semantic Web and Databases, Co-located with VLDB 2003*, Humboldt-Universität, Berlin, Germany, 7.-8. September 2003.

11. P. Dourish and V. Bellotti. Awareness and Coordination in Shared Workspaces. In *Proceedings of ACM CSCW 92 Conference on Computer-Supported Work*, pages 107–114, 1992.
12. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
13. Graham Hamilton. *JavaBeans (Version 1.01-A)*. Sun Microsystems, Inc., 8. August 1997.
14. 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*, CEUR Workshop Proceedings, 1994.
15. H. Knublauch, M. Sedlmayr, and T. Rose. Design Patterns for the Implementation of Constraints on JavaBeans. In *Tagungsband Net.Object Days 2000, Erfurt, 9.-12. Oktober*. tranSIT GmbH, 2000.
16. Mira Mezini, Linda Seiter, and Karl Lieberherr. Component integration with plugable composite adapters. In *Software Architectures and Component Technology*. Kluwer, 2000.
17. 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.
18. Joaquin Miller and Jishnu Mukerji. MDA Guide Version 1.0.1. Technical Report omg/2003-06-01, OMG, 12th June 2003.
19. C.S. Peirce. *Collected Papers of Charles Sanders Peirce*. Harvard University Press, Cambridge, 1931.
20. Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4):334–350, 2001.
21. Joachim W. Schmidt and Hans-Werner Sehring. Conceptual Content Modeling and Management: The Rationale of an Asset Language. In Manfred Broy and Alexandre V. Zamulin, editors, *Perspectives of System Informatics, 5th International Andrei Ershov Memorial Conference, PSI 2003*, volume 2890 of *Lecture Notes in Computer Science*, pages 469–493. Springer, July 2003.
22. 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.
23. Hans-Werner Sehring. Report on an Asset Definition, Query, and Manipulation Language. Version 1.0. Technical report, Software Systems Department, Technical University Hamburg-Harburg, Germany, 2003.
24. German Shegalov, Michael Gillmann, and Gerhard Weikum. XML-enabled workflow management for e-services across heterogeneous platforms. *VLDB Journal*, 10(1):91–103, 2001.
25. John F. Sowa. *Knowledge Representation, Logical, Philosophical, and Computational Foundations*. Brooks/Cole, Thomson Learning, 2000.
26. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.
27. Homepage of the Warburg Electronic Library. <http://www.welib.de>, 2004.
28. G. Wiederhold. Mediators in the Architecture of Future Information Systems. *IEEE Computer*, 25:38–49, 1992.
29. E. Yu. *Modelling Strategic Relationships for Process Reengineering*. PhD thesis, University of Toronto, 1995.