Ontology-Based Domain-Driven Design

Pavel Hruby
Microsoft
Frydenlunds Allé 6
2950 Vedbaek, Denmark
+45 29229183

phruby@acm.org

ABSTRACT

This paper suggests a method for designing domain-specific models of software applications, based on the use of domain ontologies. We will illustrate that the application objects of a domain-specific component can be derived from ontological categories for the domain and the application behavior can be modeled as aspects that cross-cut the application objects. The method leads to a specific architecture of the software component with models in two orthogonal dimensions. One dimension represents the categories originating in the domain ontology and the other dimension represents the functional concerns that originate from user requirements. Software applications consist of an interconnected set of components, and the domain ontology determines the component interfaces.

Categories and Subject Descriptors

Object-oriented Programming, Domain-specific architectures, Domain engineering, Patterns and Data abstraction

General Terms

Design

Keywords

Ontology, REA, Aspect-Oriented Programming, Business Applications.

1. INTRODUCTION

One of the most common tasks in software design is to create an object model of the software application. In designing domain-specific software, the designer has option of using knowledge about the domain, in addition to user requirements and principles of object-oriented design.

We will illustrate that using formally specified domain knowledge in software design leads to a specific structure of the model. The model addresses two orthogonal concerns; the general concepts and categories originating in the domain knowledge and the specific behavior originating in user requirements.

In this document, **domain-driven development** is understood to be the development of software applications in the scope of a specific domain or an application area. Examples of the domains are; the business domain, interactive systems domain and sales domain, see Figure 2. We call the entities of the application model **application objects**. Examples of application objects in the business domain are the sales order, invoice and shipment. The ontological **categories** correspond to the metaclasses of the application objects, the entities describing characteristics of the application objects. Examples of ontological categories in business domain are economic resource, event, agent, claim, contract and commitment. **Instances** of application objects are the runtime manifestations of application objects.

Figure 1 illustrates the application object Cash, which has a metaclass Economic Resource (an ontological category). At runtime, the instance of Cash is an object that represents, for example, the real cash contained in a wallet.
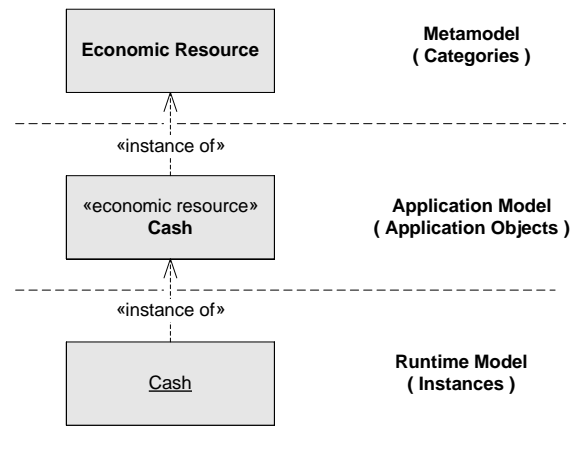

Figure 1. Category, Application Object and Instance

The term **aspect** is used to denote a concept for a module of functionality that cross-cuts application objects. There are several approaches for modeling and implementing the cross-cutting concerns, such as aspect-oriented programming, composition filters, multidimensional separation of concerns and UML collaborations. In this paper we do not discuss any of these approaches specifically, and we stay at the conceptual level.

The rest of the paper is structured as follows: the next section outlines the method of modeling the domain component, and the following sections describe each step of the method. The section on system level architecture illustrates that the software application consists of domain components with ontology-based interfaces. The last section answers some of the questions the author received while working on the paper.

2. THE METHOD

The method consists of fours steps.

1. Determine the domain. This step determines the scope of the application or a product family; examples of domains with various scopes are illustrated in Figure 2.

2. Choose an ontology for the domain. Sometimes a suitable ontology does not exist and must be developed; a metamodel for the application can play a role of the domain ontology, provided it is based on understanding of the domain principles beyond those immediately present in user requirements. An ontology should contain the minimal set of concepts that completely covers the domain.

3. Address specific user requirements. There are often requirements for functionality that cannot be part of the ontology because it is not needed in all applications in the domain. We will show that models for these requirements often cross-cut the domain objects, and can be modeled as aspects.

4. Construct the application model by configuring the objects that originate from ontological categories with aspects that originate from specific user requirements.

These four steps are described in detail in the following sections.

3. DETERMINE THE DOMAIN

Figure 2 illustrates application objects at various levels of abstraction; the horizontal dimension illustrates the domains the application objects can be applied to. General modeling languages such as UML, describing the model in terms of objects, classes and methods, contain very few semantics about real world concepts. On the other hand, their modeling scope is a domain of all object-oriented systems. Specific modeling languages that use entities such as invoice and shipment contain detailed and exact semantics and information about intentions of the model. This information can be used, for example, to validate the model against domain rules and automatically translate this model to the models in other domains. The trade-off is that the area of

applicability of a specific language is more restricted than the area of applicability of a general modeling language.

In the ideal case the modeling entities should be exactly at the level of abstraction that entirely covers the domain of the software application. We will show in the next section that the domain objects at the right level of abstraction correspond to ontological categories for the domain. For example, for the business domain, such objects correspond to categories specified by the REA (Resources, Events Agents) ontology [4, 5].
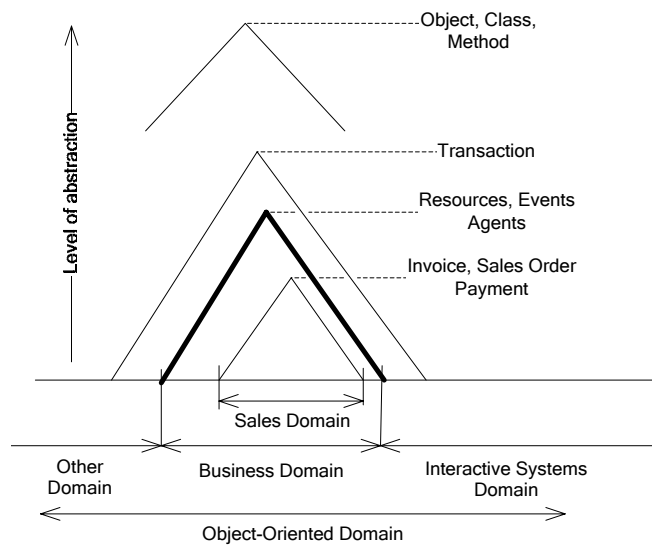

Figure 2. Levels of Abstraction and Covered Domain

4. DOMAIN ONTOLOGIES

"An ontology is an explicit specification of a conceptualization" [7]. Ontological categories define the concepts that exist in the domain, as well as relationships between these concepts. For object-oriented applications, domain ontology defines the metamodel for application models in this domain. Ontological categories correspond to the metaclasses for application objects. For example, the economic agent specified by the REA ontology is a metaclass for objects such as customer and vendor. Economic event is a metaclass for objects such as sale and payment receipt. Economic resource is a metaclass for objects such as money and item. Likewise, relationships between ontological categories become metarelationships for the relationships between application object, such as stockflow is a metarelationship for relationships called outflow and inflow; duality between economic events is a metarelationship for the reconciliation relationship between Sale and Payment Receipt.
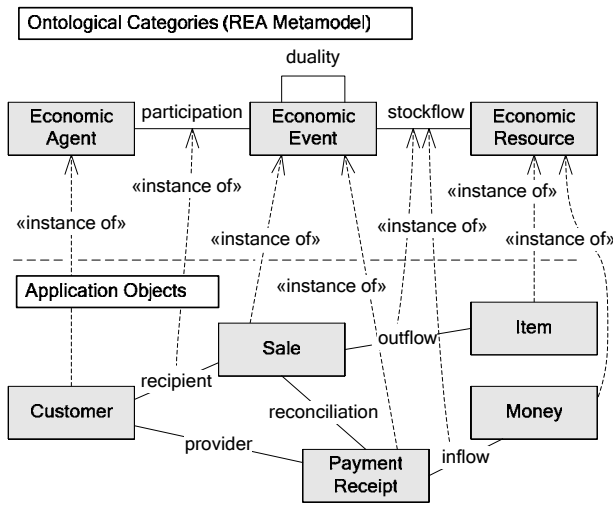
Figure 3. Application Model and its REA Metamodel

5. SPECIFIC REQUIREMENTS

5.1 Functionality of Domain Objects

The previous section illustrated that structure of a software application can be derived from ontological categories that apply to the application's domain. However, to build a useful software application, the mere structure of domain objects is not sufficient. Domain objects need functionality that is often not specified by the ontological categories, but is required by the application's users. For example, the REA ontology does not specify how to determine the identity of business objects, or how to create financial reports. However, functionality such as serial numbers and accounts are essential for the users of business applications.

Application functionality is not specified by domain ontologies for a good reason. Domain ontologies specify the structure of concepts that can be applied to all systems in the domain. Domain ontologies attempt to find the minimal, yet complete set of concepts covering the domain.

The functionality of application objects usually differs from one system to another because of specific user requirements, local conventions, as well as several other reasons. For example in business applications, some objects need human readable serial numbers, such as customers and products; some do not, such as order lines. Financial reporting depends on local legislation, lines of business and reporting usually varies from one application to another, reflecting the fact that every company is somehow different than the other. A complete list of functionality of the domain objects probably cannot be specified in general for the whole domain; users of software applications would always need new features or new versions of existing features, which cannot be foreseen by those who create the ontology.

5.2 Cross-Cutting Domain Objects

Ontological categories determine one dimension of decomposition of the domain. The other dimension of decomposition is the application functionality. In the following paragraph we will show that in many cases the modules of application functionality are not localizable into a single application object.

In the business domain, for example, the serial number of an item is an attribute of the item object. The serial number is usually not a random number. The item serial number is chosen from a number series, which is an attribute of a group of the economic resources, to which the number series is applied. Thus, the object representing the item group contains rules specifying things such as the format of the serial number, whether a serial number should be unique, how does it depend on previous numbers of the series, rules determining whether serial numbers of deleted items can be reused, and other similar rules. The number series module cross-cuts two domain objects, the item object and the item group object, and the number is constructed by mutual collaboration between the part that resides on the item and the part that resides on the item group. It is useful to think of the number series as a single module, but this module cross-cuts two application objects.
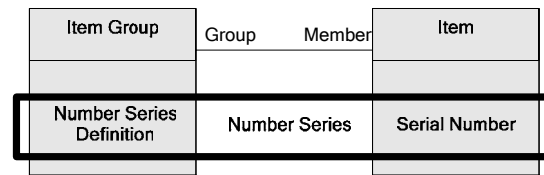

Figure 4. Number Series Cross-Cuts Application Objects

Aspect-oriented programming [10] is one of the approaches, and an addictive convention of thought on how to deal with cross-cutting concerns in a modular way. In the scope of domain-driven development, it is useful to think about entities derived from ontological categories as objects and about functionality of the software applications as aspects.

This separation of concerns also determines a mechanism of how to add the new features to the software application without changing its fundamental structure. The objects corresponding to ontological categories determine the fundamental structure of the software application and aspects provide the specific functionality.

5.3 Aspect Categories

In the section about domain objects we have shown that ontological categories correspond to metaclasses of the application objects. A similar approach can be also applied to the entities in aspect dimension.

In this section we describe metaclasses for application aspects. For example, we have shown that the number series is an aspect in the application model. However, the fundamental purpose of the number series is to give the application object unique identity.

Therefore, we can think of the number series as a specific instance of a more general aspect category called identification. Other instances of the identification aspect category are the name, phone number, e-mail address, URL (Uniform Resource Locator), GUID (Globally Unique Identifier) and ISBN (International Standard Book Number).

The identification aspect category specifies a concept of giving identity to something. Identity is not inherently part of the objects and things. People often refer to the real or imaginary things by their names. As the names are not necessarily unique within the application scope, the things are given numbers. Generally, real and imaginary things have one or more given identifiers, so that they can be referred to by using their identifiers.
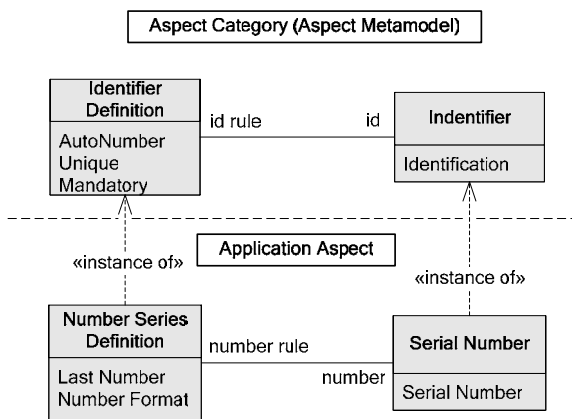

Figure 5. Application Aspect and its Metamodel

Figure 5 illustrates the identification aspect category at the aspect metaclass level and its instantiation in the number series aspect in the application model. Below we describe the identification aspect in more detail to illustrate how the aspect category is specified.

The identification aspect category at the metaclass level consists of two parts: **Identifier Definition** which defines the name of a type of identification. Identifier Type has the following attributes: **AutoNumber** - a Boolean function that indicates whether the identifier can be automatically generated by the system or not. **Unique** is a Boolean function that indicates whether the identifier is required to be unique or not. **Mandatory** is a Boolean function that indicates whether the identifier must be defined or can be undefined.

The **Identifier** part of the aspect category specifies the data type of the identification, such as a string or a number.

The application level contains aspect parts in which the parameters of the aspect categories have been set. For example, **Number Series Definition** is an instance of the Identifier Type that has automatically generated numbers. The numbers are unique and mandatory. The **Serial Number** (an instance of the Identifier) contains attributes for storing the last used number in the series and specifies the identification format; that allows the Serial Number be a combination of numbers and characters.

Other examples of aspect categories in the business domain are the address, account and posting. Their details, along with other aspect categories, have been described as behavioral business patterns [8].

The purpose of the **address aspect category**, see Figure 6, is to specify geographical locations of objects, as well as navigable routes between the locations. The address aspect has four components; the start and destination locations, the actual locations which determines the actual location of some application object and the route, which keeps track of the historical changes of the actual location. The start and destination location are usually configured on economic agents, the route is usually configured on an economic event and the actual location is usually configured on a resource.
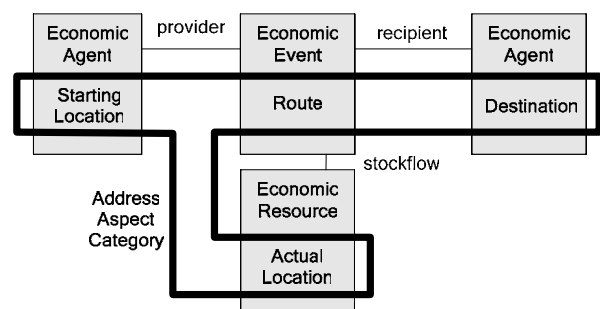

Figure 6. Address Aspect Category

The purpose of the **posting aspect category**, see Figure 7, is to keep track of transactions represented by changes of some application objects. The components of the posting aspect are the entry, which persists the application object and makes it immutable and a number of dimensions that describe the information to be registered with each entry. The entry is typically configured on an economic event or commitment and the dimensions are configured on economic agents, resources, their types and groups.
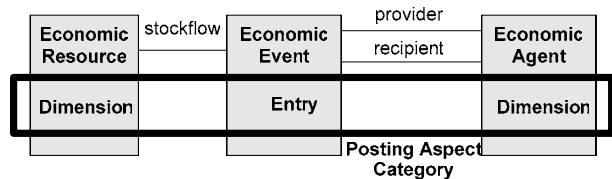

Figure 7. Entry Aspect Category

The purpose of the **account aspect category**, see Figure 8, is to represent aggregated data about entries. The components of the account aspect are the account, which represents the aggregated amount, and one or more entries, which represent the values that increase or decrease the total amount. The account is usually configured on an agent or resource type, and the entries on economic events and commitments.
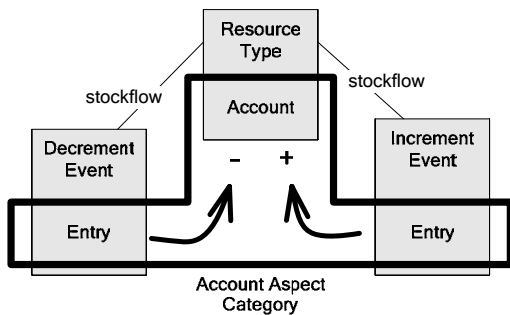
-4-

Figure 8. Account Aspect Category

5.4 Number of Aspect Categories

According to our experience the number of aspect categories for a domain is roughly at the same level as the number of ontological categories. For example, the latest version of the REA ontology [5] describes 37 ontological categories; 23 corresponding to metaclasses and 14 to metarelationships. A functionality of the CRM (customer relationship management) business application can be fully covered by 18 aspect categories.

6. CONFIGURE THE APPLICATION

We have shown that a domain-specific model can be decomposed along two dimensions: the object dimension that reflects the ontological categories of the domain and the aspect dimension that reflects the behavior, which the software application must have in order to be useful, see Figure 9. We have also shown that the components both in the object dimension and the aspect dimension can be specified at two levels of abstraction; the level of ontological categories or aspect patterns, and the level of application objects and application aspects.
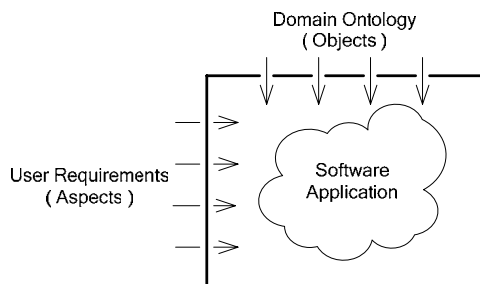

Figure 9. Objects, Aspects and Domain-Driven Development

Figure 9 illustrates the key message of this paper, which is, using domain ontologies to determine the application object model leads to a software architecture with two orthogonal dimensions.

We illustrate how this fundamental idea can be used to develop a software component in a specific domain.

If a software provider develops several applications as part of a product line, or develops multitude of very different applications, which all belong to one domain, some form of reuse of the common functionality is desirable. The architecture described in this paper allows for implementing the generalized application functionality in the aspect categories, and develop software applications by configuring the aspect categories with application objects.

The **configured application model** is a model of software application that conforms to the ontology for the particular domain and also contains specific functionality that meets user's needs. As the application objects are determined by the domain ontology, the process of creating an application model consists of assigning application aspects to application objects. This process is outlined in Figure 10.
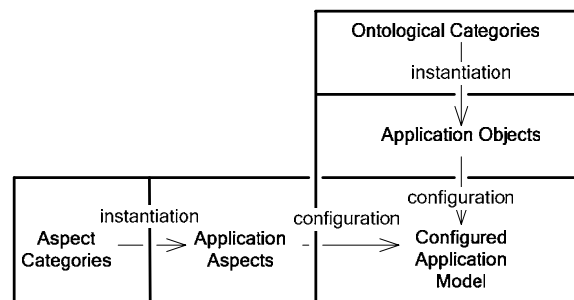

Figure 10. Application Configuration

An example of a business application configured in this way is illustrated in Figure 11. This application is a model of a simple sales module.

The ontological categories in this application are Economic Agent, Economic Event and Economic Resource; their instances are application objects Customer, Sale, Payment, Item and Cash.

The aspect categories in this application are Identification, Account, Address and Posting. Instances of the Identification aspect are Name, Item Number, Customer Number and Transaction ID. Instances of the Account aspect are Inventory Account, Bank Account, Customer Account and Cash Account. Instances of the Address aspect are Billing Address and Shipping Address. Instances of the Posting aspect are G/L (General Ledger) Entry and Inventory Entry.

The choice of the aspect categories is determined by user's needs. Other configurations of the sales process in the software applications for different users would contain a different set of aspect categories.

The Configured Application Model illustrated in Figure 11 contains the Application Objects with Application Aspects. The Customer object contains the aspects Name and Number (identification aspects), Customer Account (account aspect), Billing Address and Shipping Address (address aspects). The Sales object contains the aspects Transaction ID (identification aspect) and G/L Entry (posting aspect). Payment Receipt contains the aspects Transaction ID and G/L Entry (posting aspect). The Item object contains the Item Number (identification aspect), and the Money object contains the Bank Account and Cash Account aspects.

It has been noted that aspects typically cross-cut two or more application objects. For the sake of simplicity, the cross-cutting is not shown in the model in Figure 11, because some parts of the illustrated aspects would reside on the objects that are not shown in the model of the configured software application in Figure 11. For example, Identification Type (the other part of the Identification aspect) would be present in the objects Customer Type, Sale Type, Payment Receipt Type and Item Type. The other part of Billing Address would be present in the Invoice object; the other part of Shipping Address would be present in the Shipment Object. These objects are not illustrated in Figure 11, as the model would become too large and would obscure the main idea we want to describe in this paper.
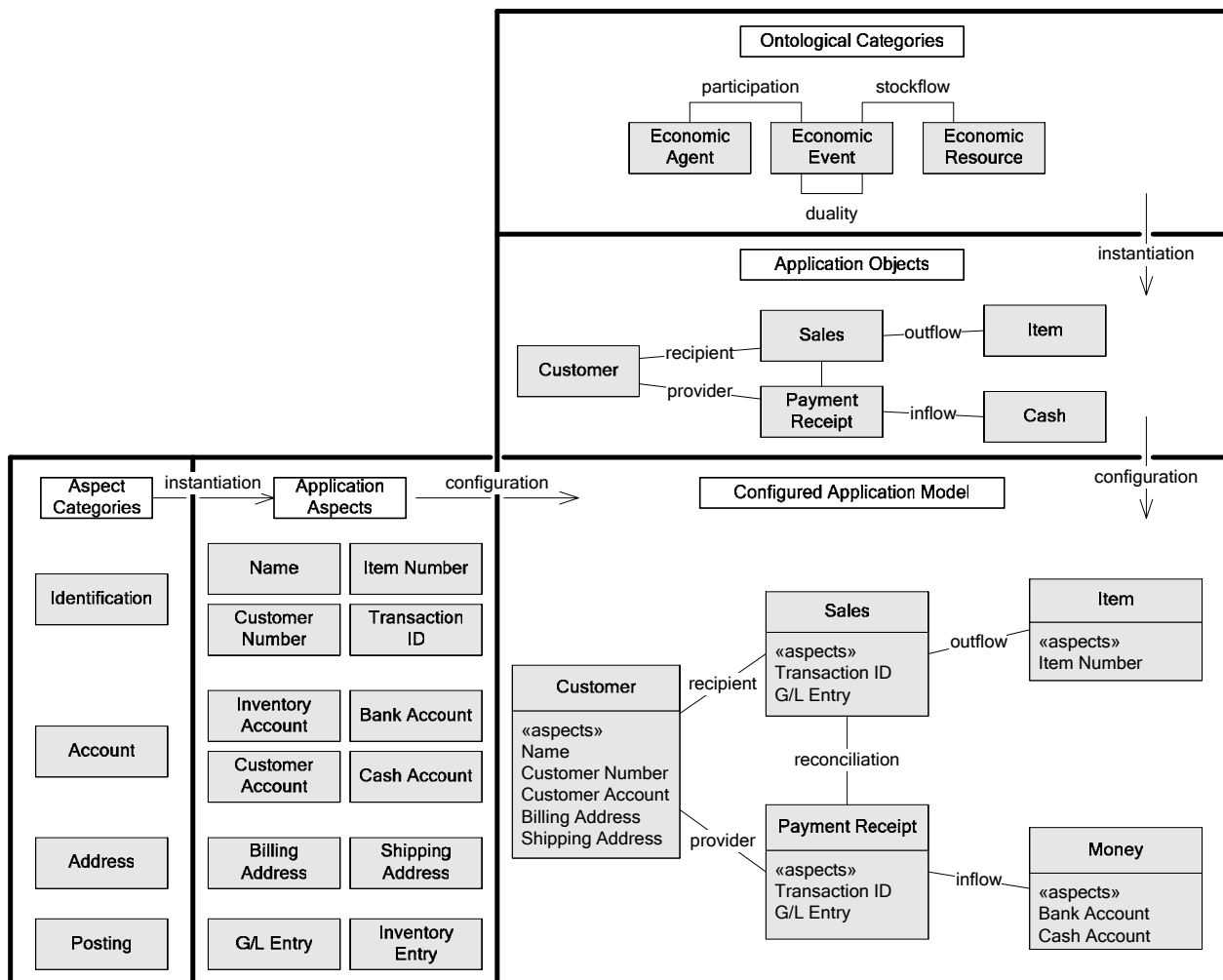

Figure 11. Example of Application Configuration

7. SYSTEM LEVEL ARCHITECTURE

This approach leads to the architecture of a domain-specific system consisting of interconnected components. There is a lower limit determining the minimal size of the component. This limit is derived from domain rules of the ontology. There is no upper limit on the number of interconnected components other than implementation technology. The rest of this section describes it in more detail.

As the ontology designers' aim is to define a minimal set of concepts that describe the domain, there usually is a small number of domain object categories. If an ontology defines domain rules, it is possible to determine a minimal set of domain object categories that must be present in the system in order to satisfy the domain rules. For example, the minimal component in the REA ontology has to include at least one increment and one decrement economic event, each of them having a relationship to an economic resource and each event having two relationships to economic agents. Figure 12 illustrates an example of a minimal REA component; if some of the object categories or relationships are missing, the design would violate the domain rules specified by the ontology.
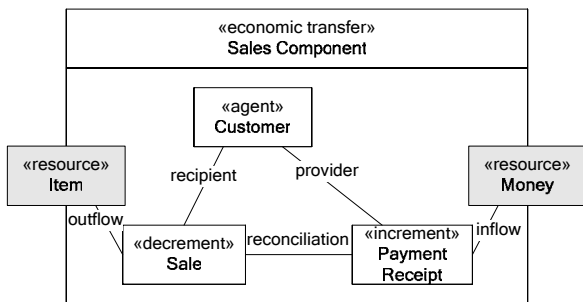
Figure 12. An example of a minimal REA component

The provided and required interfaces of the components, in Figure 12 we use the UML 2.0 port notation [12], are well defined by the shared domain ontology. For example, the ports of the REA components are the economic resources. The economic resources relate the components together into the value chain of the enterprise, see Figure 13.

Each of the components schematically illustrated in Figure 13 has its own, potentially different set of aspects. These components can be implemented as modules of a single ERP system or as several collaborating ERP systems. The components can also be replaced by legacy applications, as long as their ports can be expressed at semantic level in terms of categories of the domain ontology
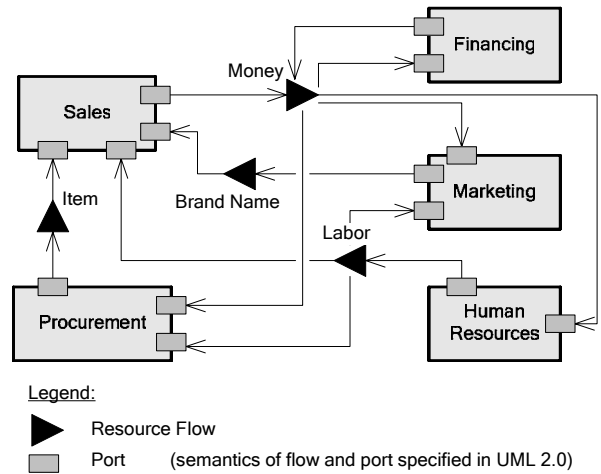
Figure 13. A system of components

8. IMPLEMENTATION

A software product based on this architecture was developed by Navision Software during 1999-2002. This development proved the feasibility of the concepts illustrated in this paper. Some of these concepts are present in Microsoft Business Framework that is being developed by Microsoft since 2002 and is intended to be part of future versions of Microsoft Visual Studio.

8.1 Composition Problem

All approaches that address configuring cross-cutting concerns must also answer the question what happens if two or more aspects try to modify the same data on an application object. The simplest approach (and best in my opinion) is to avoid that this problem ever occurs. This means that aspect categories must be designed to be non-overlapping and independent on each other, and all conform to the same interface that enables them to communicate. For example, the framework infrastructure can implement the broker pattern [2] for communication between independent aspects.

Consequence of the simple approach with non-overlapping aspects is that, for example, if a phone number is used both to contact the customer, and to identify the customer, the phone number field is configured on the customer object twice: once as a part of the address aspect and once as a part of the identification aspect. The configured application can keep these two fields synchronized. The data duplication is a trade off for simplicity and for avoiding the composition problem. Conceptually, this is all right because these two fields have different semantics for the end user.

If this simple approach is not desirable, and the software architecture allows for overlapping aspects, the composition problem occurs and must be addressed in some way. The simplest approach in this case would be to leave this problem to

the application designer who would determine the execution order of the overlapping aspects.

8.2 Object-Oriented Frameworks and Domain-Specific Languages

As the purpose and functionality of the aspect categories and ontological categories can be well defined for a given domain, it is possible to create object-oriented frameworks that implement these categories and expose these categories in the form of an domain-specific application language. In such cases, the software development of domain-specific applications can be reduced to identifying the application objects and aspects and decorating the application objects with application aspects.

A positive consequence of such approach, in which the modeling language exactly covers the domain, is that amount of code necessary to develop a software application is significantly reduced (compared to the approach in which a more general language than the language that exactly covers the domain is used).

The trade-offs are that the domain specific language is applicable only to the restricted domain and most or all application functionality is implemented in the framework, which is beyond the influence of application developers. A possible compromise is to allow the coexistence of the domain-specific language and a general language in the framework development interface, so that the requirements beyond the scope of the framework functionality can be implemented in the general language.

8.3 The Configured Model is Executable

The configured model, along with the models for aspect categories, application aspects, application objects and ontological categories, contain all information necessary to run the model. There are several ways of creating an executable application from the configured model; the detailed description is out of scope of this paper. The model can be compiled into executable software, a code can be generated from the model, then compiled and executed [1]. The model can also be run by a framework interpreting this model, or traditionally, the model can be used as a specification from which a software application is developed manually.

9. QUESTIONS AND ANSWERS

This section suggests answers to the questions the author received while working on this paper.

9.1 What Advantages does this Approach Offer in Practice?

The main advantage has already been mentioned in the introduction and discussed throughout the paper – better design of domain-specific software applications that can be verified for conformance to domain rules. However, this approach offers other interesting benefits that have potential to fundamentally change the way domain software is developed.

Transformations between domains. If transformation rules between ontologies in different domains have been described, the application objects in one domain can automatically be transformed to application objects in other domains. For example, a user interface (presentation logic domain) can be automatically derived from REA (business logic domain). Similar transformations exist between business logic, reporting and security. These functional modules are usually developed manually, but using this approach they can be created automatically using the transformation rules for the ontologies.

Verified design. If the categories of objects and aspects are implemented in object-oriented framework, the domain knowledge is available to the software application and can be used in many different ways. For example, a wizard can guide the developer to use and configure application objects to conform to the ontology. A consistency checker can verify the design against domain rules. A modeling tool can offer implementations of application objects for domain categories out of the box. Modelers can use them as they are or configure them by adding or removing application aspects.

Robust design. The separation of concerns between objects based on ontological categories and aspects based on specific user requirements, determine which parts of the system are stable and which are likely to change in time. The ontological categories won't change as long as the system stays within the boundaries of the domain. The aspects and configured application objects are subject to changing user requirements.

Component-based development. The domain ontology defines precisely what a component is in each domain, as well as the semantic interfaces of the components. This means that the "size" and interfaces of a component are no longer determined ad-hoc by developer's intuition, but can be derived from the ontology.

Defined dominant decomposition. The usual methods and techniques for aspect-oriented development, such as Theme [3] depend on the developer's intuition for what features will be modeled and implemented as objects and what features as aspects. The discussions on OOPSLA 2002 workshop on generative techniques [6] showed that the answer to this question is generally unclear. This approach gives a precise answer: objects are the entities that come from a domain ontology, the rest of the entities reside in the aspect dimension.

9.2 How come Customer Object is from the Ontology, but the Name and Address come from an Aspect (in Figure 11)?

This is because the designers of the REA ontology we have used in this example have made the clever decision not to include the categories for Name and Address in the ontology. The ontology specifies what all systems in the domain have in common. Aspects specify the features of the software applications that vary.

All systems in business domain must implement one or more entities for an economic agent, in the sales component it is Customer. However, the REA ontology specifies nothing about Names or Addresses. Indeed, not all entities must have Name and Address. For example, sales order, shipment and payment receipt do not usually have names, they are usually identified by serial numbers. Economic resources such as copyrights or stocks do not usually have addresses. The word "usually" is important, as the users might decide otherwise and applications should be able to support such requirements.

This question is also answered at more general level in sections 3.1, 4.1 and 6 of this paper.

9.3 Is Application Designer Allowed to Add Attributes to Domain Objects (in Figure 3)?

Or, should the attributes stem only from the aspects, no matter what? The in this approach, the aspects have two purposes. They capture cross-cutting concerns, as has been discussed in section 3, and they also capture the semantics of the application object's attributes. Therefore, in this approach all attributes of application objects stem from the aspects, or are accessible to the aspects. If there would be an attribute that is not part of the aspect or accessible by an aspect, the semantics of this attribute will be unknown to the business application. As a consequence, this approach forces the application developer to discover the semantic of each attribute that is required by the users, and implement an aspect category for that attribute that captures this semantic.

10. NEW EMERGING PARADIGM IN OBJECT-ORIENTED THINKING

The domain-driven design of software applications seems to push the boundaries of current object-oriented thinking. Some of the questions, asked by experts in object-oriented technology after reading this paper, indicate that we talk about a new paradigm and the transition to domain-driven development might be difficult for people who are used to the traditional object-oriented approach.

We have illustrated that in the domain-driven development of software applications, application designers can use domain ontologies as one of the sources for creating application models, in addition to traditional analysis based on user requirements. The conceptualized domain knowledge in the form of domain ontology can be used as a metamodel for application models.

However, domain ontologies cannot describe specific functionality and the differences between different applications in a given domain, which originate in user requirements, because the concepts of the domain ontology must be applicable to all systems in the domain. The modules of functionality resulting from the user requirements are often not localizable into application objects that originate in the domain ontology.

To solve this conflict, we have illustrated that the domain-specific software applications lead to the architecture of a component with two orthogonal dimensions. The object dimension represents the categories originating in the domain ontology and the aspect dimension represents the functional modules that originate from user requirements.

This approach determines the software architecture for domain specific software applications, allowing customizations within a given domain and adding new features into the existing components without changing their fundamental structure. As this approach specifies the semantics of ports of the components and subsequently the component connectors, it also determines the software architecture at the system level.

11. REFERENCES

[1] Czarnecki, K. Eisenecker, U.W*: Generative Programming - Methods, Tools, and Applications*, Addison-Wesley, 2000

[2] Buschmann, F. et al: *Pattern-Oriented Software Architecture,* Wiley, 1996

[3] Clarke, S, Baniassad, E.: *Theme: An Approach for Aspect-Oriented Analysis and Design, ICSE 2004*

[4] Geerts, G. McCarthy, W. *The Ontological Foundation of REA Enterprise Information System*s, Michigan State University, August 2000

[5] Geerts, G. McCarthy, W. *An Ontological Analysis of the Economic Primitives of the Extended REA Enterprise Information Architecture*, The International Journal of Accounting Information Systems, 2002, Vol. 3, pp. 1-16

[6] Generative Techniques in the context of Model Driven Architecture, Workshop at OOPSLA 2002

[7] Gruber, T. R. *A Translation Approach to Portable Ontology Specifications*, Knowledge Acquisition, 1993.

[8] Hruby, P. *Universal Enterprise Model*, in: VikingPloP 2002, Proceedings, Microsoft Business Solutions, 2002.

[9] Hruby, P. et al: *Business Patterns*, Springer-Verlag, to be published.

[10] Kiczales, G. et al: *Aspect-Oriented Programming*, in: M. Aksit and S. Matsuoka (Eds.): ECOOP '97 Proceedings, Jyväskylä, Finland, June 1997, Springer-Verlag Berlin Heidelberg 1997.

[11] Tarr, P. Ossher, H. Harrison, W, Sutton, S. M. Jr. *N Degrees of Separation: Multi-Dimensional Separation of Concerns,* Proceedings 21st International Conference on Software Engineering (ICSE'99), May 1999.

[12] *UML 2.0 Superstructure Specification*, OMG document ptc/04-10-02, 2004.