FORMALIZATION & DATA ABSTRACTION DURING USE CASE MODELING IN OBJECT ORIENTED ANALYSIS & DESIGN

Meena Sharma and Rajeev G Vishwakarma

IET, Devi Ahilya University, Indore, India
SVITS, Rajiv Gandhi Technical University of MP, Indore, India
meena@myself.com, rajeev@mail.com

ABSTRACT

In object oriented analysis and design, use cases represent the things of value that the system performs for its actors in UML and unified process. Use cases are not functions or features. They allow us to get behavioral abstraction of the system to be. The purpose of the behavioral abstraction is to get to the heart of what a system must do, we must first focus on who (or what) will use it, or be used by it. After we do this, we look at what the system must do for those users in order to do something useful. That is what exactly we expect from the use cases as the behavioral abstraction. Apart from this fact use cases are the poor candidates for the data abstraction. Rather the do not have data abstraction. The main reason is it shows or describes the sequence of events or actions performed by the actor or use case, it does not take data in to account. As we know in earlier stages of the development we believe in 'what' rather than 'how'. 'What' does not need to include data whereas 'how' depicts the data. As use case moves around 'what' only we are not able to extract the data. So in order to incorporate data in use cases one must feel the need of data at the initial stages of the development. We have developed the technique to integrate data in to the uses cases. This paper is regarding our investigations to take care of data during early stages of the software development. The collected abstraction of data helps in the analysis and then assist in forming the attributes of the candidate classes. This makes sure that we will not miss any attribute that is required in the abstracted behavior using use cases. Formalization adds to the accuracy of the data abstraction. We have investigated object constraint language to perform better data abstraction during analysis & design in unified paradigm. In this paper we have presented our research regarding early stage data abstraction and its formalization.

KEYWORDS

Behavior, Data, Abstraction, Software Development, Use Case Model, Use Case, Modeling, UML, Formalization, OCL

1. INTRODUCTION

Use case modeling is meant for capturing requirements. It can include several UML (Unified Modeling Language) diagrams to model the requirements. UML is a standardized general-

David C. Wyld (Eds) : ICCSEA, SPPR, CSIA, WimoA - 2013
pp. 67–75, 2013. © CS & IT-CSCP 2013 DOI : 10.5121/csit.2013.3507

purpose modeling language in the field of object-oriented software engineering. The standard is managed, and was created, by the Object Management Group. It was first added to the list of OMG adopted technologies in 1997, and has since become the industry standard for modeling software-intensive systems. In software and systems engineering, a use case is a list of steps, typically defining interactions between a role (known in UML as an "actor") and a system, to achieve a goal. The actor can be a human or an external system. Use cases represent the things of value that the system performs for its actors. Use cases are not functions or features, and they cannot be decomposed. Use cases have a name and a brief description. They also have detailed descriptions that are essentially, stories about how the actors use the system to do something they consider important, and what the system does to satisfy these needs. In computer science, abstraction is the process by which data and programs are defined with a representation similar in form to its meaning (semantics), while hiding away the implementation details. Abstraction tries to reduce and factor out details so that the programmer can focus on a few concepts at a time. A system can have several abstraction layers whereby different meanings and amounts of detail are exposed to the programmer. For example, low-level abstraction layers expose details of the computer hardware where the program is run, while high-level layers deal with the business logic of the program. We will investigate abstraction of data in use cases.

2. USE CASE MODELING

Use case modeling is to capture the system (to be) requirements making use of UML diagrams like- Use case diagram, class diagram, activity diagram, sequence diagram and state-chart diagram. It may further include other diagram as and when required. A use-case model is a model of how different types of users interact with the system to solve a problem. As such, it describes the goals of the users, the interactions between the users and the system, and the required behavior of the system in satisfying these goals. A use-case model consists of a number of model elements. The most important model elements are: use cases, actors and the relationships between them. A use-case diagram is used to graphically depict a subset of the model to simplify communications. There will typically be several use-case diagrams associated with a given model, each showing a subset of the model elements relevant for a particular purpose. The same model element may be shown on several use-case diagrams, but each instance must be consistent. If tools are used to maintain the use-case model, this consistency constraint is automated so that any changes to the model element (changing the name for example) will be automatically reflected on every use-case diagram that shows that element. The use-case model may contain packages that are used to structure the model to simplify analysis, communications, navigation, development, maintenance and planning. Much of the use-case model is in fact textual, with the text captured in the use-case specifications that are associated with each use-case model element. These specifications describe the flow of events of the use case. The use-case model serves as a unifying thread throughout system development. It is used as the primary specification of the functional requirements for the system, as the basis for analysis and design, as an input to iteration planning, as the basis of defining test cases and as the basis for user documentation.

3. ABSTRACTION AND DATA ABSTRACTION

Abstraction captures only those details about an object that are relevant to the current perspective. The concept originated by analogy with abstraction in mathematics. The mathematical technique of abstraction begins with mathematical definitions, making it a more technical approach than the general concept of abstraction in philosophy. For example, in both computing and in

mathematics, numbers are concepts in the programming languages, as founded in mathematics. Implementation details depend on the hardware and software, but this is not a restriction because the computing concept of number is still based on the mathematical concept. Data abstraction refers to, providing only essential information to the outside word and hiding their background details i.e. representing the needed information in program without presenting the details. Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation. Let's take one real life example of a TV which you can turn on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players BUT you do not know it's internal detail that is, you do not know how it receives signals over the air or through a cable, how it translates them, and finally displays them on the screen. Thus we can say, a television clearly separates its internal implementation from its external interface and you can play with its interfaces like the power button, channel changer, and volume control without having zero knowledge of its internals. Now if we talk in terms of C++ Programming, C++ classes provides great level of data abstraction. They provide sufficient public methods to the outside world to play with the functionality of the object and to manipulate object data ie. state without actually knowing how class has been implemented internally. Abstraction is the process of recognizing and focusing on important characteristics of a situation or object and leaving/filtering out the un-wanted characteristics of that situation or object. So, you see that Abstraction is the basis for software development. It’s through abstraction we define the essential aspects of a system. The process of identifying the abstractions for a given system is called as Modeling (or object modeling). So, we see that abstraction is the basis for object oriented programming. Abstraction serves as the foundation for determining the classes for a particular system (which is called object model). But be advised, there is no acid test to decide if the abstraction for a given system is right or wrong. A "person" abstraction for a hospital information system would be different from a person abstraction for a library information system and even with hospital information system, person abstraction may be different for different projects. Once you have abstracted an object, it can be re-used. It can be modified to suit other situations. As a child you learnt Tri-cycle. You used the experience of learning tri-cycle (handle bar control, pedaling) to learn bicycling. What actually you do to learn bicycling is that you only learn to balance the bicycle while you use the experience of tricycle to use handlebar and pedaling. The same case applies to abstraction as well. Though abstraction seems to be a simple concept, it’s a challenging task. The reasons are

1. There are un-limited numbers of possibilities to define an abstraction for a situation.
2. As mentioned earlier, there is no acid test to determine if the abstraction is right or wrong.

You end up discussing, arguing with your counterpart that yours is best and his is worst…. He does the same thing…

However, these problems are always addressed as you gain experience (which you can gain by reading more books/articles and doing real time projects) in defining the abstraction. Abstraction by itself is a huge and an interesting concept. But, we feel that most of the people, who define

4. EXAMPLE – HOW TO ABSTRACT DATA

It is an important step to abstract data. Data is abstracted from the steps involved in describing the behavior. Use case is made up of such steps. In each step of event we abstract data. An example is shown as below.

Table 1. Data Abstraction Example

Activity- " Open the door"	
Behavioral Steps	**Data Abstraction**
Walk till the door	Distance - meters
Catch the knob of the door	Force value - Kilograms
Swing the knob clockwise	Angle of swing in degrees
Push the door	Direction of swing
	Door Size – meter x meter
	Force of Push - kilograms

The above example gives the details of the data abstraction for a simple task- 'open the door' in real life. The steps involved can be seen as the actions performed to fulfill a use case named- Open the Door. The right column depicts the data to be abstracted. First of all to reach the door one has to walk some distance- distance walked is the data associated with this step. Further when the person reaches the door he catches the knob of the door. In order to catch the knob one needs some force to be applied on the door.

5. CASE STUDY INVESTIGATIONS – E-RETAIL SYSTEM

Unified process and UML have special representations for use cases and actors. A simple ellipse represents use cases; simple stick figures, actors. A use case diagram expresses the system's use cases in relation to the actors that invoke them. We prefer to construct these diagrams with an adornment that draws a boundary around the system, placing the use cases within the boundary and the actors outside Figure 1 as shown below. This notation is convenient when multiple systems are being jointly developed, and it can be clearly seen which use cases are part of which system. The complete collection of use cases, actors, and diagrams forms a use case model, which, like individual use cases, is just one part of the system's requirements specification.

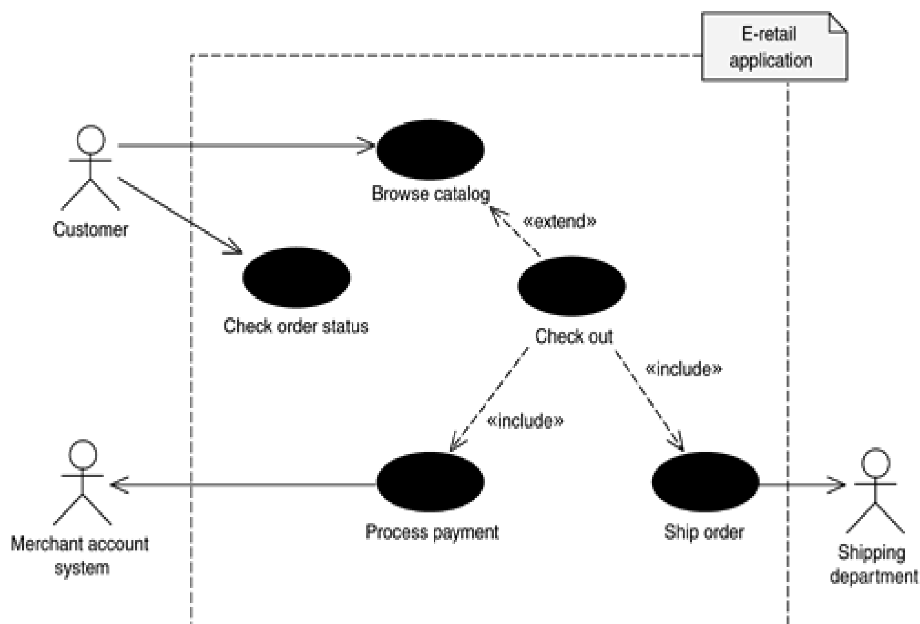

Fig. 1. Use Case Model for E-Retail System

Relationships between actors and use cases indicate that the actor can invoke a particular use case. Figure 8-4 shows that the actor can browse the catalog or check on the order status. This diagram also identifies two other actors: the Merchant Account System and the Shipping Department. These actors are in fact not individuals but rather external systems, or more accurately the APIs of those external systems. When the Process Payment use case is invoked, it initiates the dialogue between the Merchant Account System and this system; similarly, the Ship Order use case contacts the Shipping Department's system when invoked. Relationships between use cases are also documented in the model and appear in a diagram as a stereotyped dependency relationship, or arrow-headed dashed line. The two major types of relationships between use cases are «includes» and «extends». Both indicate a dependency on one use case by the other; however, the subtle difference between the two has, in my experience, caused a significant degree of confusion. In Figure 1, the use case Checkout extends the use case Browse Catalog. This means that the actor invoking the use case might decide to extend the dialogue with the system to include the activities described in the Checkout use case. The key word here is "might." Every invocation of the Browse Catalog use case does not necessarily include checking out. The relationships between the Checkout use case and the Process Payment and Ship Order use cases, however, are expected with every invocation of the Checkout use case. The «includes» relationship indicates that each invocation of the Checkout use case will invoke the included use cases at least once. This type of a relationship can be thought of as calling a subroutine. The use case model shows the structural relationships between use cases but does not show dynamic relationships, or workflows. Dynamic behaviors are expressed with interaction diagrams: sequence, collaboration, and activity. It is important to keep that in mind when drawing use case diagrams. It's easy to start constructing use case diagrams as if they were workflow diagrams, since use case names often reflect major activities of the business workflow. Most associations between use cases in a diagram simply imply that one invokes the other, nothing more. There's more to the use case model than this high-level diagram. Some analysts recently have begun to create a single activity diagram for each use case in the system. This activity diagram helps me sort out all the alternative flows in a use case and ensure that we haven't forgotten how any of them rejoins the basic flow. In the past, we drafted a few use cases in which later iterations added alternative flows that seemed reasonable and addressed the issue at the time, but when it came to implementing them, it wasn't clear in what state those alternative flows would leave the system and at what point the normal flow of the use case scenario should resume. Properly created activity diagrams for a use case clearly identify all alternative paths as branches and even allow us to see the effects of repetition, or loops, in the flow that would otherwise be mentioned only in the text of the use case specification. Figure 2, the activities for the Browse Catalog use case, shows a typical activity diagram for a use case that has some repetition. It is clear from this diagram that there is one main loop structure, labeled the Shop controller. Many of the branches are guarded with conditions—text between the brackets—and indicate the availability of an option that the actor can invoke. As a notational convention, we can add notes to the activity diagram at those points where the use case is extended or included. If the modeling tool you use supports the inclusion of hyperlinks to other diagrams, it is highly recommended, hyperlinking these notes to the relevant activity diagram to make it easier for the reader to follow the flow through all the related use cases. The activity diagram of a use case clarifies the nature of the «includes» or «extends» relationships. In the case of an «extended» relationship, it should be possible for the actor to get from the start activity to an end activity without invoking the extended use case. Figure 2 demonstrates that it is possible for the actor to follow a path from start to finish without hitting the Request Checkout activity and thus invoking the Checkout use case.

6. INTRODUCING FORMALISM

We have suggested and investigated that Object Constraint Language (OCL) is very helpful to gain accuracy and perfection. The Object Constraint Language is a declarative language for describing rules that apply to Unified Modeling Language models developed at IBM and now part of the UML standard. Initially, OCL was only a formal specification language extension to UML. OCL may now be used with any Meta-Object Facility (MOF) Object Management Group (OMG) meta-model, including UML. As an example, we have modeled a computer system for a fictional company called Royal and Loyal (R&L). R&L handles loyalty programs for companies that offer their customers various kinds of bonuses. Often, the extras take

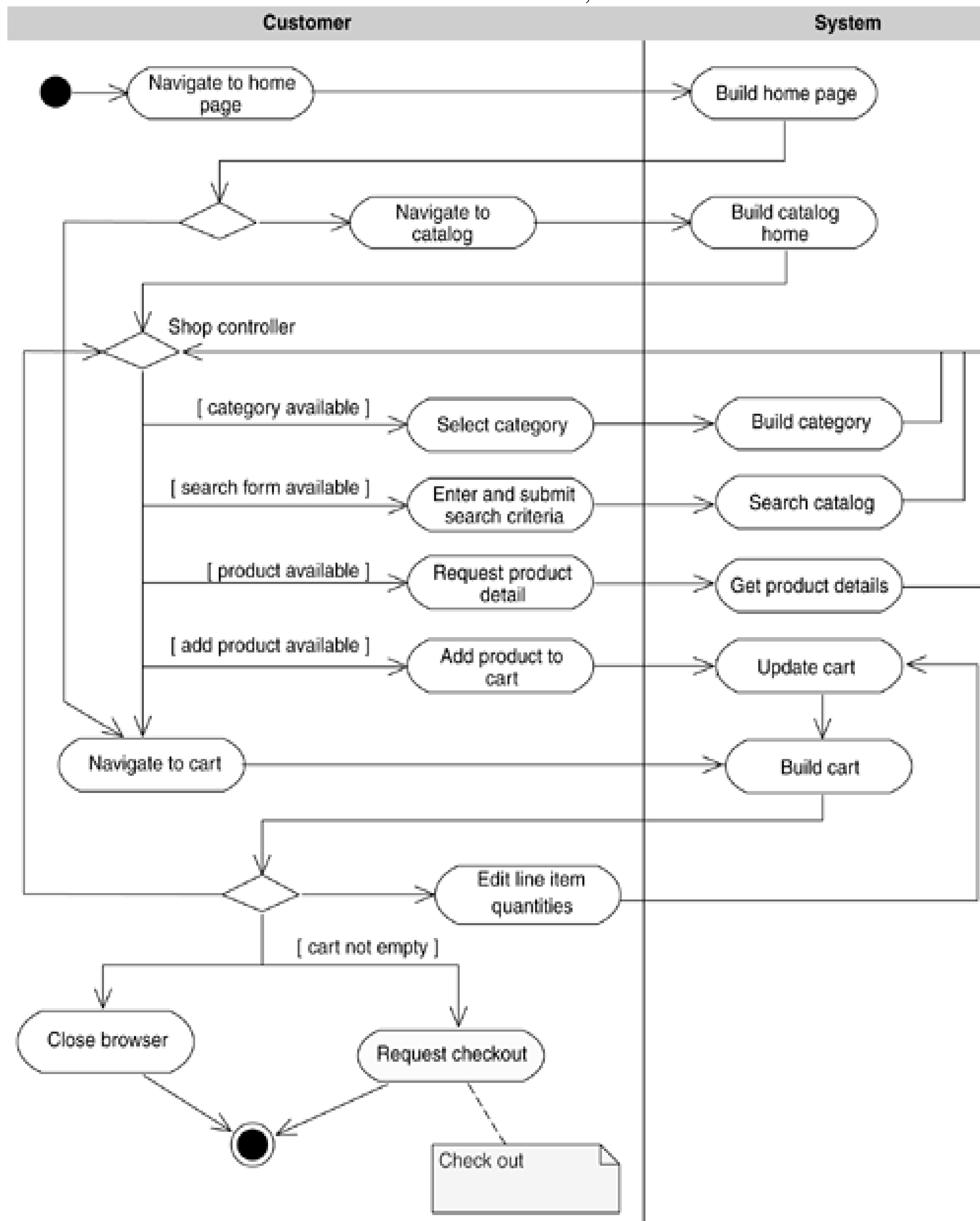

Fig. 2. The figure shows the activity diagram for Browse Catalog Use Case

the form of bonus points or air miles, but other bonuses are possible as well: reduced rates, a larger rental car for the same price as a standard rental car, extra or better service on an airline, and so on. Anything a company is willing to offer can be a service rendered in a loyalty program. The central class in the model is LoyaltyProgram. A system that administers a single loyalty program will contain only one instance of this class. In the case of R&L, many instances of this class will be present in the system. A company that offers its customers membership in a loyalty program is called a ProgramPartner. More than one company can enter the same program. In that case, customers who enter the loyalty program can profit from services rendered by any of the participating companies. Every customer of a program partner can enter the loyalty program by filling in a form and obtaining a membership card. The objects of class Customer represent the persons who have entered the program. The membership card, represented by the class CustomerCard, is issued to one person. Card use is not checked, so a single card could be used for an entire family or business. Most loyalty programs allow customers to save bonus points. Each individual program partner decides when and how many bonus points are allotted for a certain purchase. Saved bonus points can be used to "buy" specific services from one of the program partners. To account for the bonus points that are saved by a customer, every membership can be associated with a LoyaltyAccount.

Various transactions on this account are possible. For example, the loyalty program "Silver and Gold" has four program partners: a supermarket, a line of gas stations, a car rental service, and an airline.

- At the supermarket, the customer can use bonus points to purchase items. The customer earns five bonus points for any regular purchase over the amount of $25.
- The gas stations offer a discount of 5 percent on every purchase.
- The car rental service offers 20 bonus points for every $100 spent.
- Customers can save bonus points for free flights with the airline company. For every flight that is paid for normally, the airline offers one bonus point for each 15 miles of flight.

In this situation, there are two types of transactions. First, there are transactions in which the customer obtains bonus points. In the model, these transactions are represented by a subclass of Transaction called Earning. Second, there are transactions in which the customer spends bonus points. In the model, they are represented by instances of the Burning subclass of Transaction. The gas stations offer simple discounts but do not offer or accept bonus points. Because the turnover generated by the customers need to be recorded, this is entered as two simultaneous transactions on the LoyaltyAccount, one Earning and one Burning for the same number of points.

Customers in the Silver and Gold program who make extensive use of the membership are rewarded with a higher level of service: the gold card. In addition to the regular services, customers who have a gold card are also offered the following additional services:

- Every two months, the supermarket offers an item that is completely free. The average value of the item is $25.
- The gas stations offer a discount of 10 percent on every purchase.
- The car rental service offers a larger car for the same price.
- The airline offers its gold card customers a business class seat for the economy class price.

Customers must meet at least one of the following conditions to get a gold card:

- Three sequential years of membership with an average annual turnover of $5,000
- One year of membership with a turnover of $15,000, where the turnover is the total turnover with all program partners

To administer different levels of service, the class ServiceLevel is introduced in the model. A service level is defined by the loyalty program and used for each membership.

R&L advertises the program and its conditions. It manages all customer data and transactions on the loyalty accounts. For this purpose, the program partners must inform R&L of all transactions on loyalty program membership cards. Each year, R&L sends new membership cards to all customers. When appropriate, R&L upgrades a membership card to a gold card. In this case, R&L sends the customer a new gold card along with information about the additional services offered, and R&L invalidates the old membership card.

The customer can withdraw from the program by sending a withdrawal form to R&L. Any remaining bonus points are canceled and the card is invalidated. R&L can invalidate a membership when the customer has not used the membership card for a certain period. For the Silver and Gold program, this period is one year. Initial values and derivation rules will be as follows.

```
context LoyaltyAccount::points
init: 0
context CustomerCard::valid
init: true
context CustomerCard::printedName
derive: owner.title.concat(' ').concat(owner.name)
```

In this case, the refined query operation can be specified as follows:

```
context LoyaltyProgram::getServices(pp: ProgramPartner)
                                : Set(Service)
body: if partners->includes(pp)
    then pp.deliveredServices
    else Set
    endif
```

7. SUMMARY

We see that there is great importance of data abstraction in early stages of life cycle. The process involves identifying use cases and then after finding out actors and use cases, we describe the events performed by the system and the actor. This is called the flow of events required to complete a uses case basic flow or any other use case scenario. Inside the flow of events, there is hidden data. Actually in these set of flow of events we expect behavior performed by the interaction of actor and the system. Behavior is the key to find out or abstract data. Now in order to find out data we carefully understand the flow of event step by step and abstract data. Formalization contributes to the accurateness of the data abstraction. We have examined and explored object constraint language to perform improved data abstraction during analysis &

design in unified paradigm. In this paper we have presented our research regarding early stage data abstraction and its formalization.

REFERENCES

[1] Beck, Kent: Extreme Programming Explained: Embrace Change. Addison-Wesley, Boston, 2000.
[2] Block, Robert: The Politics of Projects. Yourdon Press, New York, NY, 1983.
[3] BridgePoint Object Action Language Manual www.projtech.com/pdfs/bp/oal.pdf
[4] BridgePoint Object Action Language www.projtech.com/pdfs/bp/oal.pdf
[5] Cockburn, Alistair: Writing Effective Use Cases. Addison-Wesley, Boston, 2001.
[6] Communications of the ACM, Volume 44, Issue 10. (The entire issue is devoted to aspect-oriented development.)
[7] Constantine, Larry, and Lucy Lockwood: Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design. Addison-Wesley, Reading Mass., >1999.
[8] Fowler, Martin: "Use and Abuse Cases." In Distributed Computing, April 1998.
[9] Harel, D. "Statecharts: A visual formalism for complex systems." In Science of Computer Programming 8, 3 (June 1987), 231–274.
[10] Jacobson, Ivar, et al.: Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, Reading, Mass., 1992.
[11] Kabira Technologies URL: www.kabira.com
[12] Lamport, Leslie, and P. M. Melliar-Smith. "Synchronizing clocks in the presence of faults," in Journal of the ACM, 32(1):52–78, January 1985.
[13] Mellor,Stephen J., Steve Tockey, Rodolphe Arthaud, and Philippe Leblanc: Software-Platform-Independent, Precise Action Specifications for UML. In Proceedings of «UML 98».
[14] Page-Jones, Meilir: Fundamentals of Object-Oriented Design in UML. Addison-Wesley, Boston, 2000.
[15] Palahniuk, Chuck: Fight Club. WW Norton and Company, 1996. "The first rule of Fight Club is you do not talk about Fight Club. The second rule of Fight Club is you do not talk about Fight Club."
[16] Project Technology, Inc. URL: www.projtech.com
[17] Rumbaugh, James: The Unified Modeling Language Reference Manual. Addison-Wesley, Reading MA, 1998
[18] Shlaer, Sally, and Stephen J. Mellor: Object Lifecycles: Modeling the World in States. Prentice Hall PTR, Englewood Cliffs, N.J., 1992
[19] Shlaer, Sally, and Stephen J. Mellor: Object Lifecycles: Modeling the World in States. Prentice Hall PTR , Englewood Cliffs, N.J., 1992.
[20] Shlaer, Sally, and Stephen J. Mellor: Object-Oriented Systems Analysis: Modeling the World in Data. Prentice-