January 1998

The shared model : the key to analysis without paralysis

Lois M. L. Delcambre

Earl F. Jr Ecklund

The Shared Model: The Key to Analysis without Paralysis

by

Lois M. L. Delcambre
Computer Science and Engineering Department
Oregon Graduate Institute
Portland, Oregon 97291-1000
lmd@cse.ogi.edu

Earl F. Ecklund, Jr.
*OBJECT*ive Technology Group
Beaverton, Oregon 97008, USA
ecklund@teleport.com

Abstract

The concept of the shared model is introduced in this paper to direct the analysis process. The shared model is a mental model in the mind of the various users of the system; it corresponds to their understanding of what the software system does. The shared model is manifested first in the use case model which describes how the system functions and how it manipulates various domain concepts/objects, from the point of view of the user. The shared model is elaborated in the analysis object model, consisting of objects and responsibilities that support the use cases. The shared model provides clear guidance about when to stop the analysis "chase". We believe that the analysis step is critical to the development of a software system because it identifies the cumulative demand on the domain objects and because it enables reuse of objects and responsibilities at the domain level. One of the key challenges in analysis is deciding when to stop, to resist the temptation to forge ahead with decisions that rightly belong in design. The shared model provides the key to deciding what is in and what is out of both the use case and the analysis models.

1. Introduction

The analysis step of object-oriented software development is often maligned, perhaps with good reason. Analysis is the process of discovering what the users expect the system to do. Analysis documents what the user will see, what the user will do, and the what the user expects the system to do in response. In our work, the analysis step consists of developing the use case model and developing the analysis-level object model, based on use case-driven CRC sessions. Analysis coincides with active user participation; the users must be able to confirm the use cases, including any factored use cases. The users must also be able to understand and confirm the analysis-level object model and the sequence diagrams that support the use cases.

The most difficult challenge during analysis is to decide when and how to stop. During a CRC session, every time you assign a responsibility to an object you ask yourself, "Can this object handle this responsibility on its own? Or does it need a collaborator?" We need specific criteria to make this decision. In general terms, we want to avoid premature design; we don't want to unintentionally constrain the design space. We also want to take the benefit of the analysis step without spending any extra (a.k.a. wasted) time. How do we strike this balance?

We define the shared model as the key to maximizing the success of the analysis step. The *shared model* is a mental model, in the mind of the users, that embodies the user's understanding of the software system. The shared model is "shared" between the users and the software system. The shared model must include the concepts or objects that the user requires for the system to represent. The shared model must describe the user interactions that must be possible.

And the shared model must prescribe the operations that the user will invoke. The description of an operation in the shared model includes the response from the system, in terms that the user understands, in addition to the details of how the operation is requested. Our job during analysis is to elicit and manifest the shared model in the analysis work products: the use case model and the analysis-level object model.

In this paper, we first explain the importance of analysis, using work products excerpted from the analysis of a system that tracks the grades for a university course. The next two sections describe the impact of the shared model on the development of the use case model and on the analysis-level object model. The paper concludes with a discussion of related work followed by conclusions.

2. The Importance of Analysis

Analysis consists of the activities that follow the requirements specification. In the sense of Jackson, requirements are necessarily about the application domain and not about the "machine" [Jackson95]. In analysis, and in the shared model, we focus entirely on the users understanding of what takes place inside the machine. Neither the shared model nor the analysis step is focused on the real world. Neither produces a model of the real world. Rather we place our attention on that subset of the real world which is necessarily projected onto the machine. Requirements gathering, of the style recommended by Jackson, would precede our analysis activities. We advocate analysis because it provides the critical connection between what the users need from the software system (to be built) and the actual construction of the software system.

Analysis activities precede the design and implementation of the system. We see the demarcation between analysis and design as the place where the implementation and deployment environments are considered. Design is where the components of the systems to be constructed are decided and where performance tradeoffs are considered.[1] Analysis is, therefore, *not* concerned with these things. More than that, analysis must not limit the possible construction options. Premature design (done unintentionally during analysis) is most often bad design. Premature design results in bad design because the full complement of demands to be placed on the software are not yet known nor prioritized and because the implementation and deployment options are also not yet being considered. If you can think of two reasonable designs and the analysis model precludes one of them, then it is a bad analysis model.

So why bother with analysis?

Analysis, through the articulation of the shared model, makes it clear what the system must do. Requirements need not give the complete story, need not communicate the shared model. Requirements fill a different role; they often set the contractual basis for system acceptance. Analysis, through the shared model, allows the developers and the users to work together to describe the system under construction. Analysis comprises the steps where user participation and user review are essential.

The manifestation of the shared model occurs at two levels of detail: first, at the more abstract, less detailed level in the use case model and second, at the fully elaborated level in the analysis

---

[1] We advocate an iterative and incremental approach to software development. For each iteration, a given slice of the requirements or use cases would follow through the steps of use case model development, analysis, design, and implementation.

object model. The shared model thus provides guidance for the development of the use case model and for the analysis object model.

The analysis-level object model is the first object model. We translate the user-visible functions, as described in the use case model, into responsibilities of objects. A use-case-driven approach to CRC, much like [Wirfs-Brock90], results in the initial choice of objects. Responsibilities are assigned to objects precisely because they are needed to support the use cases. This provides a direct connection between the use case model and the actual objects that appear in the software system. The objects selected during analysis are precisely the objects that appear in the shared model**.** The analysis session focuses on the discovery of entity objects, i.e., domain objects, that are part of the shared model and the proper crafting of responsibilities for these objects.

Analysis provides another critical benefit: the reuse of domain-level objects and responsibilities. Using CRC, every time an object is needed we have the chance to identify and reuse an existing object, i.e., a card that is already on the table. Similarly, when we use an existing object, we have the opportunity to reuse, and possibly generalize, an existing responsibility. This is the most critical and the most valuable part of the analysis process. With diligence and insight, the analysis-level object model identifies reuse. And this type of reuse, at the domain or user-visible level, is generally not identified at later steps of the software development. If two responsibilities or two objects are described in different ways, then it is very likely that they will both be designed and implemented. To do less, would be to shirk the responsibilities of the design and implementation. But during analysis, the second or subsequent time that an object or responsibility is used represents one step towards a cleaner object model.

We describe the analysis-level object model using class names and responsibility names, expressed in natural language [Wirfs-Brock90]. We also document the purpose of each class and the purpose of each responsibility. It is through the statement of purpose that we have the chance to identify opportunities for reuse. It is also through the refinement of the statement of purpose that we can adjust a class or a responsibility to enable reuse.

Consider an example taken from a pension planning system that allows an employee to setup his or her retirement process. The retirement benefit of an employee may be split between the employee and a divorced spouse, if the divorce settlement includes a Qualified Domestic Relations Order (QDRO). Suppose that the first use case to be analyzed allows an employee to plan for retirement. One of the objects that appears in the shared model is the pension payment plan. When used for this first use case, a pension payment plan must be established for the employee, the person who is retiring. When the subsequent use case handling a QDRO is analyzed, we discover that a pension payment plan must also be established for the domestic partner (usually an ex-spouse). By doing analysis using the CRC technique and considering the earlier responsibility to establish a pension payment plan (implicitly for the employee), we discover the opportunity to generalize the responsibility to establish a pension payment plan for the person indicated by the parameter. This may or may not have been anticipated by the analyst when the responsibility was first discovered. But either way, the analysis process can lead to reuse of responsibilities. The details of the QDRO example showing how analysis of anticipated functionality expressed as change cases can improve the quality of an analysis model, can be found elsewhere [Ecklund96].

It is through the statement and refinement of the purpose that we discover reuse. If there is an object or responsibility for which there is disagreement about its purpose, then this raises a red flag [Tonge97]. The users and the developers must be clear about the purpose of each object and each responsibility. A disagreement may signal that the current object embodies two different objects, in fact.

We demonstrate our analysis work products through an example, taken from the use case model and the analysis-level object model for a system that tracks grades for a university course. We have selected three use cases: Manage Teams, Assess Team Contribution, and Select Work Item or Component, shown in Figures 1, 2, and 3. The grader system defines a work item to be the unit of work that is turned in to be graded. A work item might be a single homework assignment, a test, or one installment of a project. Each work item may consists of components and the components may have subcomponents. For example, a homework assignment might be assigned in two parts, where the first part consists of five questions and the second part consists of three questions. The homework work item would have two components, with five and three (leaf-level) subcomponents each. The grader system allows individual work items to be declared to be team projects and the composition of teams to be different for each work item. (The composition of teams tend to remain the same, but may change because of students dropping the course or because of personality or schedule conflicts, for example.) The Manage Teams use case allows teams to be defined and adjusted for a given work item, for any work item that was declared to be a team project. The Assess Team Contribution use case allows individual students to provide a frank, confidential description of the activities and contributions of the members of a team. This can be used, with discretion, by the instructor in the evaluation of borderline grades when final averages are computed. The Select Work Item or Component use case is a use case that has been factored out because it is used by more than one other use case. In particular, it is used by the Assess Team Contribution and the Manage Teams use cases to establish the proper work item. It is also used in other use cases, e.g., to throw out a component by setting the weight in the grading policy to zero for the relevant leaf-level components. (Sometimes a particular question on a homework assignment or test is judged, by the instructor, to be misleading or ambiguous. The instructor can choose to throw the component out: to compute grades as if the question didn’t exist.)

Use Case 109. Manage teams

Actor: Instructor (or grader, if delegated to grader)
Purpose: This use case allows teams to be defined or adjusted. Each team is associated with one or more work items, reflecting the team that submitted the work item.
Preconditions: Instructor (or grader) has been validated (and grader has been delegated this function). The course has been selected. The manage teams use case has been invoked.
Assertions: Each team consists of students that are valid students for this class. Teams are associated only with valid work items for this class. Each student belongs to one and only team for each team-graded work item.

Scenario: New teams can be created by entering the work item(s) that the team is associated with plus the set of students that comprise the team. The actor may select work items from a display showing all work items defined for the class. The actor may select students from a display showing all students in the class.

Existing teams can be changed (when an error was made in the original recording of teams) by changing the work item(s) associated with the team or the team composition. When adding work item(s), the actor may select work items from a display showing all work items defined for the class. When adding students to the team composition, the actor may select students from a display showing all students in the class. When removing students or work items from a team, the actor may select students or work items from a display showing all students and all work items currently on the team.

[Note, if the team composition changes, e.g., from one work item to the next, then a new team should be defined. As an example, if Smith, Jones, and Walker comprise a team for work item 1, but Smith and Walker comprise a team for work item team, then two different teams need to be defined (one for Smith, Jones and Walker and one for Smith and Walker).]

Postconditions: For each team, the new or updated team is recorded in the grader system, consisting of the requested students and associated with the requested work items.

Figure 1. Manage Teams Use Case

Use Case 106. Assess team contributions

Actor: Student
Purpose: Capture each student's assessment of other team members' contributions to each team-graded work item.
Preconditions: "Assess team" was requested from student session; class and student's userid are known, student is enrolled in class.
Assertions: Student may enter comments only for her/his team; student may only view and edit comments he/she has entered.

Scenario: Student uses use case 130 to select which team-graded work item to assess; system displays names of members of student's team; student edits text describing teamwork and assessing each team member's contributions.

Postconditions: Assessment comments are stored in system. Only instructor and this student have permission to view them.

Figure 2. Assess Team Contribution Use Case

Use Case 130. Select work item or component

Actor: Indirect, any user
Purpose: Select a work item or component as a parameter to perform some other use case function.
Preconditions: Invoked from use cases 105, 106, 107, 108 or 110. User role and class are set.
Assertions: When invoked from use case 105, only work items may be selected. When invoked from use case 106, only team-graded work items may be selected.

Scenario: From the work items and components defined for the set class, system displays work items (to student role) or all work items and components (to grader or instructor roles). User makes selection, which is forwarded to calling use case.

Postconditions: Either a work item or component is selected, or the selection is cancelled.

Figure 3. Select Work Item or Component Use Case

The CRC session driven by these use cases is captured in an analysis-level, UML sequence diagram [Fowler97b]. These sequence diagrams indicate the objects that were introduced (shown as labels on the top of the vertical lines) and the responsibilities assigned to objects (shown on the labels of arcs inbound to the vertical line. We use three type of objects based on the work of Jacobson [Jacobson92]: control objects, interface objects, and entity objects to provide a separation of concerns and robustness to change. Figures 4, 5, and 6 show the analysis-level sequence diagrams for the above three use cases, respectively.

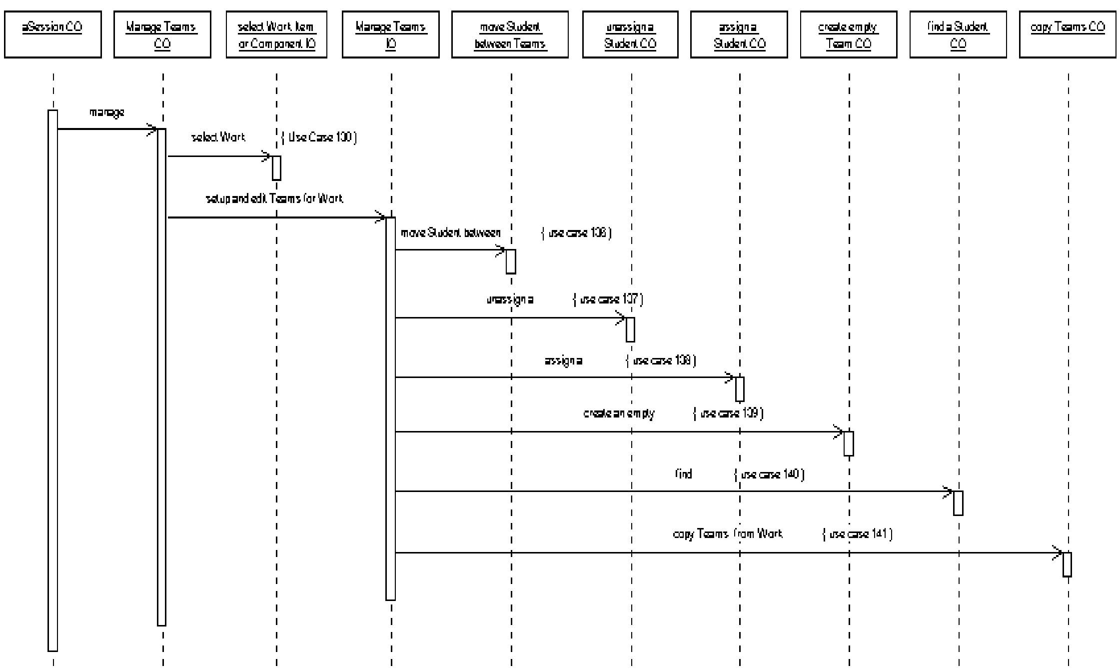

Figure 4. Manage Teams Sequence Diagram

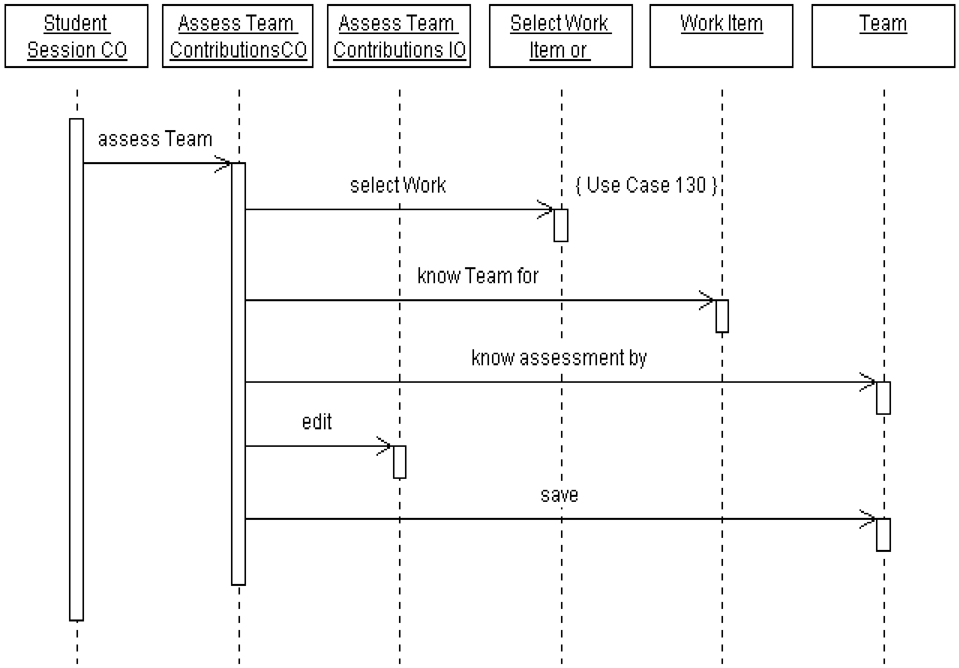

Figure 5. Assess Team Contribution Sequence Diagram

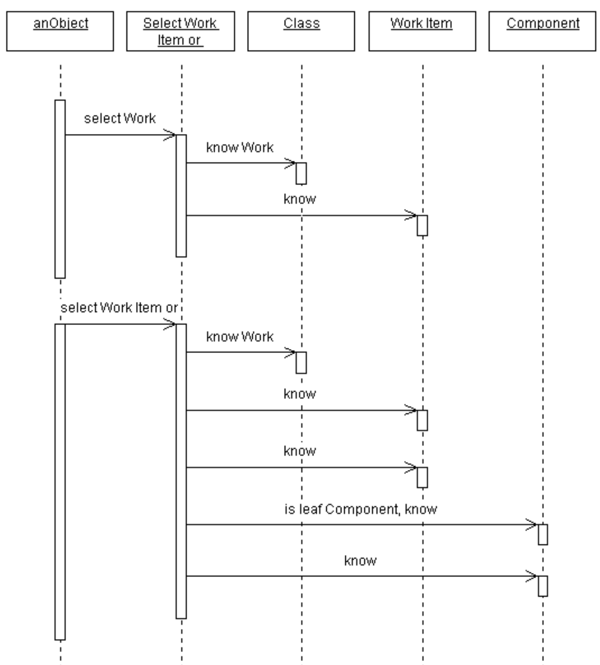

Figure 6. Select Work Item or Component Sequence Diagram

Why is it important to do analysis? Analysis allows you to discover the cumulative requirements, the cumulative responsibilities of the domain objects involved in the software system. When all of the use cases have been analyzed, the objects and their associated responsibilities suffice to support the shared model.

3. The Impact of the Shared Model on Use Case Development

We use the shared model as the litmus test for developing the use case model. The question: "Does this concept or action belong in the use case?" is answered by the question: "Is this concept or action in the shared model?" Consider the Manage Teams use case, shown in Figure 1. As described by Jacobson [Jacobson92], user interface sketches can be developed in conjunction with the use case model. One possible user interface sketch is shown in Figure 7. The purpose of the sketch is to help discover what functions are required in the shared model. For the manage teams interface sketch, six operations were discovered: create a new, empty team; move a student from the unassigned list to a team; move a student from one team to another; move a student from a team to the unassigned list; find a student by name (and indicate which team he or she is on, if any); and copy the team assignments from another work item. The details of whether student names are cut/pasted, or dragged/dropped, or selected in some other manner is not important at this stage. (These decisions should be made during interface design.) But the identification of the six operations to be supported is the key accomplishment. The six operations are part of the shared model. These six operations are therefore described as six use case extensions in the refined use case model. Notice that this interface sketch also identifies a number of domain concepts that are part of the shared model: student, class, work item, and team.

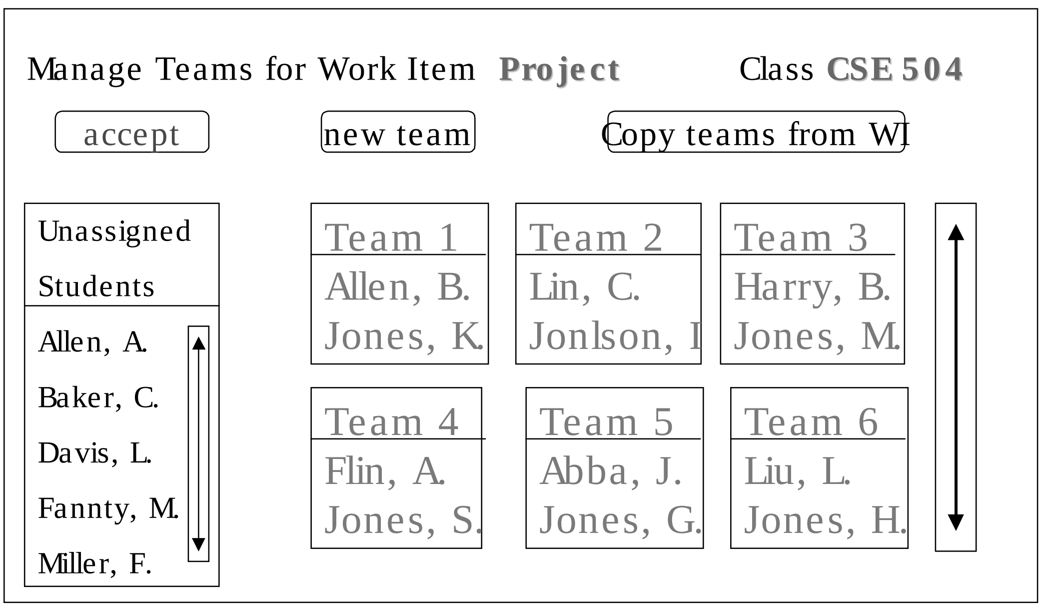

Figure 7. Interface Sketch for Manage Teams Use Case

The Manage Teams user interface sketch also demonstrates that persistence and transactional processing can be in the shared model. The interface sketch includes an "Accept" indicator, e.g., in the form of a button that the user can click on. The functioning of the system in response to the user pressing the Accept button is to commit the entire sequence of changes made to the teams on the current display, the teams for the particular work item. Similarly, the system must back out this same sequence of changes if the user does not presses the Accept button. The point here is that the transactional nature of updating team composition is clearly in the shared model, inasmuch as the buttons are present on the interface sketch. In another system, the user may have no awareness of transactions; they may be defined entirely by the system developers. In this second case, transactions are obviously not in the shared model. The shared model thus embodies precisely those concepts that the user understands. Note that all users have a shared model. Some of the users may have detailed knowledge of the domain and thus a detailed shared model. We would call such users *domain experts* and expect that they are a subset of the users of the system.

We use the concept of the shared model to help define the use case model because we believe that it is essential for the use case model to describe more than simply the shallow exchange, e.g., of characters, across the system boundary. Users always have an understanding, at various levels of sophistication, about the processing that takes place inside the software system in response to their input. The purpose of the use case model is to articulate the system behavior, at this level. This view is consistent with others who have emphasized the importance of capturing more than just the interface details in the use case model [Chandrasekaran97].

4. Impact of the Shared Model on Analysis

As mentioned in Section 2, the challenge for analysis is to prevent the analyst from making decisions that interfere with design decisions that are to follow. As an example, we wouldn't want the analyst to select objects that supported a binary search. The appropriateness of a binary search, as opposed to a linear or some other style of search, should be made at design time when we know the programming language, implementation environment, class library, the size of the data structures, the frequency of operations, etc. to be used. We must stop the analyst from going too far.

How does the shared model stop the analyst from going too far? The kind of stopping criteria that we need is when we decide whether the object can handle this responsibility, the one we have just assigned it, on its own, at this level of abstraction. This is the basic question for the CRC method. Normally, if the object can handle the responsibility by itself, then we stop. Otherwise, we look for a collaborator. It's when we look for the collaborators that we introduce new objects. We believe that it is appropriate to stop the analysis "chase" whenever the proposed collaborator is *not* in the shared model. Another way to say this is that only the domain objects that belong in the shared model are allowed to enter into the analysis-level CRC sessions.

As an example of how we stop, consider the sequence diagram of assess team contribution shown in Figure 5. We see that the use case (as supported by the AssessTeamContributionsCO) must find the team for this particular student (the user in this case) for the selected work item. The AssessTeamContributionsCO collaborates with the Work Item (the one returned by the SelectWorkItemorComponentIO object) to "know Team for Student." From prior analysis, we are aware that the work item must know all teams. And that each team is comprised of students. We thus believe that it is possible to ask the work item to find the team for this particular student. To go further would be to introduce some form of search structure or search process, to check through the various teams for the work item or to navigate from the student to the team or some such algorithm. These details are not in the shared model. We assume that a work item will be able to fulfill this responsibility and we push no further in the analysis. Note that this responsibility, to find the team for the current student user, is given to the work item object because, as part of the shared model, the user must assess the team contribution for a particular team, attached to a particular work item. It is visible to the user that he or she must first select the work item in order to assess the team.

Does the ISA relationship appear in the analysis-level object model? Only if it appears in the shared model. As an example, in the grader system we have three different human actors:

instructor, grader, and student. All human actors must login to the system and must identify the role that they are taking for a given course. The details of login are the same for all three roles; it is based on their unix userid. Because this is reflected in the use case model (not shown in this paper for the sake of brevity), this implies that the concept of a user, as an abstract superclass of instructor, grader, and student is in the shared model. Further, the use of the unix userid to identify the person is also in the shared model. (The shared model may have selected the use of the unix userid to identify person as a matter of convenience; unix userids are known by users familiar with e-mail addresses. Regardless of the underlying reason, the fact that the unix userid is mentioned in the use case is tantamount to declaring that the unix userid for all users is in the shared model.

What do analysis patterns [Fowler97a] have to do with the shared model and the analysis-level object model? Analysis patterns, as described by Fowler, provide elegant object model structures for domain objects. As such, they offer an opportunity for the analysts to refine the analysis-level object model. Such refinement should be reviewed and approved by the users, as with all other aspects of the analysis-level object model.

What do design patterns [Gamma95] have to do with the shared model? How should design patterns influence analysis? What seems like a nonsensical question is not. We believe that knowledge of design patterns can give the analyst the confidence to stop the analysis chase. As an example, knowledge of the State Pattern, clearly provides a way to support state-specific behavior for a given object. Knowing that, it suffices to describe the state-specific responsibilities, where they differ from the default responsibilities, without trying to decide how state transitions will be represented and supported. As another example, knowledge of the Composite Pattern, might allow the analysis to stop earlier when dealing with recursive component structures. We can simply assume that a component knows it subcomponents, by some means.

5. Comparison with Related Work

5.1 Separating Analysis and Design

One fundamental contribution of the Shared Model is its guidance in when and how to stop the CRC chase and thus demarcating the separation of Analysis and Design. Most authors writing on object-oriented software development (Object-Oriented Analysis and Design or Object-Oriented Software Engineering) discuss the separation of requirements, analysis, and design phases in the development process model, mostly as separations that are hard to identify, often inferring change of phase from shifts in the vocabulary. For example, Vayda states that “the dividing line between requirements modeling and early analysis is often hard to distinguish, but this causes no real problem.” [Vayda95] And Booch observes that “the boundaries between analysis and design are fuzzy, although the focus of each is quite distinct.” [Booch94]

Some practitioners appear to minimize or elide analysis, often by moving analysis-level reasoning earlier (into use cases) and/or later (into design) in the development process [e.g., Firesmith91, Jordan94, Wirfs-Brock90]. Firesmith states, “I do not make a strong barrier between analysis and logical design. I make a stronger barrier between logical design and implementation design.” Booch changed the name of [Booch94] to avoid this appearance.

Often, one sees characterizations of analysis as describing “what” the system will do, and design as describing “how” the system will do it. Høydalsvik et al assert that instead of what/how, the real difference between analysis and design should be defined in terms of

- whether the constructs used in the model are part of the problem-domain (analysis) or part of the solution (design), and
- whether the method addresses communication with the user, including the need for verification and validation. [Høydalsvik93]

In Fusion, Coleman et al also identify analysis with the problem domain and design with the implementation domain: “Separating the behavior of a system from the way it is implemented requires viewing the system from the user’s perspective rather than that of the machine. Thus analysis is focused on the domain of the problem and is concerned with externally visible behavior.” [Coleman94] This is fairly close to the shared model orientation, since the focus on the problem domain is a side effect of viewing analysis from the user’s perspective, or from the shared model.

Champeaux et al state that “Analysis models may omit mention of classes, constraints, operations, etc., that do not appear to affect the overall functionality of the system, with the expectation that these issues will be further examined downstream” [Champeaux92a]. This feels very close to the shared model perspective of analysis: one includes in the analysis model classes, operations, etc. that users perceive to affect the functionality of the shared model. (The included classes are naturally introduced as collaborators of other shared model objects during the analysis CRC chase.) Meanwhile one avoids introducing helper and other implementation objects, which are beyond the shared model stopping point, until downstream design activities.

Vayda listed six analysis modeling issues:

1. Failure to isolate analysis and design activities
2. Misuse of OO Analysis modeling concepts
3. Model granularity and scaleability issues
4. Connecting and ensuring the consistency of multiple modeling views
5. Failure to use proper tools
6. When to stop modeling?

In his discussion of issue 6, when to stop modeling, Vayda says “This simple point has caused serious problems for many projects. Analysts new to OT often feel that they must have a perfect analysis model before proceeding on to design and implementation. This ailment has been referred to as ‘analysis paralysis’” [Vayda95]. The shared model plays a strong role in pruning the CRC chase and resolving this stopping issue.

5.2 Facilitating Communication between Users and Developers

Another aspect of the shared model is its role as the common model of the system’s functionality, which further enables the analysis model to facilitate communication between users and developers. Hayes and Coleman stated that “Analysis is intended to facilitate the formulation and communication of object-oriented descriptions of the problem domain” [Hayes91]. Jordan et al sought a similar user (e.g., system engineer) – developer communication role with their “Object-Oriented Requirements” (OOR):

> The OOR should not be considered the design, but rather a conceptual model of the system. The important thing is that system engineers and developers have the same model. This eliminates the paradigm shift and is the overriding factor in easing the transition from requirements to development. On the other hand the common model does help developers think through alternative designs while reviewing requirements. [Jordan94]

The shared model is a mental model, an abstraction based on the collective perceptions of the users and domain experts, and their expectations for the system's operation. As such, the shared model cannot be validated, and it can only be extracted by interviews with multiple users, who possess their individual views of the shared model.

5.3 Mental Models

Such mental abstractions are common, and go by many names, such as the: "essential behavior" [Page89] , "essential model" [Alabiso88], "mental objects" [Beech88], "mental model" [Bos89, Fraser92, Harrison93, Hartman94, Horn92, Leathers90, Wirfs-Brock90] and "concept space" [Goldberg95]

Wirfs-Brock et al state that "mental models abstract out those features of a system required for our understanding, while ignoring irrelevant features" [Wirfs-Brock90]. Hartman et al observe that "the goal of Information Design is to create an accurate mapping between the information modeled by the system and the user's mental model" [Hartman94]. The shared model aligns with these desirable behaviors.

Goldberg and Rubin's concept space is an abstraction for "what the expert knows, believes, assumes" about analysis, and "analysis focuses on what a system is supposed to do, rather than on how it is supposed to do it" [Goldberg95]. Therefore we see that their concept space models "what the expert knows, believes, assumes [about] what a system is supposed to do", which comes as close to the shared model as any other work in the literature. One thing that is different for us is that we base the line around analysis on the shared model. That is, we base the decision to include or exclude a potential collaborator class in the analysis model (i.e., to continue or stop the chase) on whether or not the collaborator class is perceived to be in the shared model.

Fraser et al warn us that "the problems with mental models lie not in whether they are right or wrong by definition, all models are simplifications. The problems with mental models arise when the models are tacit when they exist below the level of awareness" [Fraser92]. The shared model shares this exposure to the unstated. Therefore it is very important that all models understandable to the domain experts be reviewed by them. This provides an opportunity for these users to learn from the developers' understanding of the shared model and perhaps refine the shared model. It also enables the developers to elicit unstated aspects of the shared model that become recognized as discrepancies between the users perception of the shared model and the developers version of the shared model come to light.

5.4 Mental Models of What?

Insomuch that many object-oriented analysts have a mental model of the system, we should ask, what is it the mental model is a model of? The "real world", an abstraction of the real world (the problem domain), or perhaps the implementation domain. Spinelli et al note that "a user often has a model of a system which is different from the real physical structure of the software" [Spinelli94]. Hartman et al add, "The goal of Information Design is to create an accurate mapping between the information modeled by the system and the user's mental model" [Hartman94].

There is popular folklore that an object-oriented approach makes it easier to model the real world. Aksit and Bergmans consider that "The preparatory work in the analysis phase consists of mapping between the real world entities and the entities in the analysis model: objects" [Aksit92]. They appear to be in good company, as the three amigos have all espoused to model the real world [Booch94, Jacobson86, Rumbaugh87, Rumbaugh91]. Rubin et al discuss modeling a real-time control system, where the "Current Problem Space models the state of the real controlled system" [Rubin88].

Others endorse modeling the problem domain [Hayes91, Kerth91, Seidewitz87]. These problem domain models are abstractions of the real world, and model things external to the system. Goldberg and Rubin, Jacobson et al, and Jordan et al propose conceptual models of various names: concept space [Goldberg95], conceptual picture of the system [Jacobson92], and conceptual model of the system [Jordan94]. These conceptual models (picture) seem to be models of the software. Jacobson et al and Jordan et al are referring to explicit object models of the software, perhaps their respective finished analysis and requirements models.

The shared model differs from these models, in that it is an informal model in the users' domain (not an explicit object model) of the software. Only Goldberg and Rubin's concept space share this characteristic with the shared model.

5.5 Cook and Daniels's Specification Model

Cook and Daniels describe three models [Cook94]. The first is an abstraction of the real world, a model of selected entities from the world. The other two are models of the software at two levels, which they call the specification level and the implementation level. The specification model is an abstraction of the software; it is an explicit object model expressed in the user's domain (which corresponds to our analysis model). The implementation model is a complete, explicit object model of the software expressed in the implementation domain (which corresponds to our design model). In contrast, the shared model is an informal model of the software, expressed in the user's domain; the specification model can be mapped into the shared model. If one thinks of the shared model as a window into the specification model, between the specification model and the world model, then their mapping from the specification model to the shared model can be identified with the projection of the correspondence between their specification model and the world model into the shared model window, with a particular focus on the place where the software is visible to the world

The shared model is thus not a model of the real world; it is a reflection of some portion of the real world: precisely that portion of the real world that has a representation in the software system to be built. The interactive development of the shared model, through the collaboration of domain experts with analysts, allows the relevant domain objects and behaviors to emerge. We focus on discovering the relevant domain objects and behaviors of the shared model rather than attempting to create any model of the real-world. This places our work as an activity that follows what Jackson describes as requirements [Jackson95].

Another significant difference in our work from that of Cook and Daniels and the work of Rumbaugh is that we focus in the shared model on use cases, objects, and responsibilities during analysis rather than states and events. We drive the articulation of the shared model by writing use cases and then discovering objects and their responsibilities based on the work of Wirfs-Brock [Wirfs-Brock90] using three types of objects (interface, control and entity) from Jacobson [Jacobson92]. We see the use case model as an early manifestation of the shared model which then drives the discovery of objects, responsibilities, and collaborations, the elaboration of the shared model.

One major motivation for demarcating the shared model is to emphasize the models that must be confirmed by the domain experts. Use cases, written in natural language, with the scenario describing one path through the various cases and alternatives, provides something that user can either confirm or refute. This is one of the strengths of the use case model: it does not require that the user understand programming or any form of pseudo code. Each use case can be confirmed or refuted, as written. For the analysis process, sequence diagrams that transcribe the CRC session have this same strength, since users can reason about them from their understanding of the shared model.

5.6 The Importance of Analysis

Some authors (e.g. Goldberg) focus on analysis, some (e.g. Wirfs-Brock) focus on design, and others (e.g. the three amigos) advise balanced use of the models. For example, Jacobson et al advise us that "the analysis model is the basis of the system's structure. In this model, we specify all the logical objects to be included in the system and how these are related and grouped" [Jacobson92].

Champeaux et al and Vayda et al report the following observations and lessons learned regarding the importance of a solid analysis model:

> The output of the analysis serves two different purposes. First, it provides a check on the requirements. By using another semiformal notation, we obtain a deeper insight into the development task. A client, if different from the development team, will be able to agree (or not) that the analysis output captures the best insights about what the target system is supposed to be. Secondly, the output of the analysis is channeled into the design phase [Champeaux92b].

> The most important lesson is to insist on a distinct analysis phase and to keep the analysis work products relatively free of implementation aspects. This combined with the other points … should allow a project to get the most fundamental development product, the analysis model, into good shape [Vayda95].

and

> In retrospect, it is now clear that a primary benefit of using the OO approach was the way that some of the analysis models become a powerful framework upon which the design was built [Champeaux92b].

6. Conclusions and Future Work

This paper defines the shared model, a mental model of the domain objects, interface, and operations that must be embodied in the software system to be built. The shared model provides focus for the analysis activities by setting the scope. Referring to the shared model allows us to determine whether something is in or out of the analysis work products. The shared model provides focus specifically for the development of the use case model and the analysis object model. Thus the shared model is manifest first in the use case model and then elaborated in the analysis object model.

In general we are working on the analysis phase, seeking ways to maximize the benefit of analysis without unnecessary activity or premature design. The shared model clearly contributes to this goal. We are investigating other aspects of this problem such as using the knowledge of design patterns to stop the analysis chase (with confidence) and specify only those aspects appropriate at the analysis level, such as state-specific responsibilities for use by the state pattern. We are also working on being more precise about how domain objects enter into the CRC process and how they are interconnected in the shared model and in the current context for the user.   Finally, we are interested in the traceability of the analysis model into the design model. How can we retain the integrity of the analysis model, even when design has proceeded? Generally, the initial design model is the same as the analysis model. It is then refined and expanded during design to include additional objects and to provide much more detail, such as signatures and data structures.

7. References

[Aksit92] M. Aksit and L. Bergmans, Obstacles in Object-Oriented Software Development, Proc. OOPSLA '92

[Alabiso88] B. Alabiso, Transformation of Data Flow Analysis Models to Object Oriented Design, Proc. OOPSLA '88

[Beech88] D. Beech, Intensional Concepts in an Object Database Model, Proc. OOPSLA '88

[Booch94] G. Booch, Object-Oriented Analysis and Design, 2nd Ed., Benjamin/Cummings, 1994

[Bos89] J. Van Den Bos and C. Laffra, PROCOL A Parallel Object Language with Protocols, Proc. OOPSLA '89

[Champeaux92a] D. de Champeaux, D. Lea and P. Faure, The Process Of Object-Oriented Design, Proc. OOPSLA '92

[Champeaux92b] D. de Champeaux, A. Anderson and E. Feldhousen, Case Study of Object-Oriented Software Development, Proc. OOPSLA '92

[Coleman94] D. Coleman, S. Arnold, D. Bodoff, C. Dollin, H. Gilchrist, F. Hayes and P. Jeremaes, Object-Oriented Development: the Fusion Method, Prentice Hall, 1994

[Cook94] S. Cook and J. Daniels, Designing Object Systems: object-oriented modeling with Syntropy, Prentice-Hall International, 1994

[Ecklund96] E. Ecklund, L. Delcambre and M. Freiling, "Change Cases: Use cases that identify future requirements", Proc. OOPSLA '96

[Firesmith91] D. Firesmith, Workshop: Object Oriented (Domain) Analysis, OOPSLA '91 Addendum to the Proceedings

[Fowler97a] M. Fowler, Analysis Patterns: Reusable Object Models, Addison-Wesley, 1997

[Fowler97b] M. Fowler and Scott, UML Distilled: Applying the Standard Object Modeling Language, Addison-Wesley, 1997

[Fraser92] S. Fraser, L. Marshall and T. Bailetti, Workshop: Team Approaches to OO Design, OOPSLA '92 Addendum to the Proceedings

[Gamma95] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995

[Goldberg95] A. Goldberg and K. Rubin, Succeeding with Objects, Decision Frameworks for Project Management, Addison-Wesley, 1995

[Harrison93] W. Harrison and H. Ossher, Subject-Oriented Programming (A Critique of Pure Objects), Proc. OOPSLA '93

[Hartman94] M. Hartman, F. Jewell, C. Scott and D. Thornton, Taking an Object-Oriented Methodology into the Real World, OOPSLA '94 Addendum to the Proceedings

[Hayes91] F. Hayes and D. Coleman, Coherent Models for Object-Oriented Analysis, Proc. OOPSLA '91

[Horn92] B. Horn, Constraint Patterns As a Basis For Object Oriented Programming, Proc. OOPSLA '92

[Høydalsvik93] G. Høydalsvik and G. Sindre, On the purpose of Object-Oriented Analysis, Proc. OOPSLA '93

[Jackson95] M. Jackson, Software Requirements & Specifications, Addison-Wesley, 1995

[Jacobson86] I. Jacobson , Language Support for Changeable Large Real Time Systems. Proc. OOPSLA '86

[Jacobson92] I. Jacobson, M. Christerson, P. Jonsson and G. Ostergaard, Object-Oriented Software Engineering, Addison-Wesley, 1992

[Jordan94] R. Jordan, R. Smilan and A. Wilkinson, Streamlining the Project Cycle With Object-Oriented Requirements, Proc. OOPSLA '94

[Kerth91] N. Kerth, A Structured Approach to Object-Oriented Design, OOPSLA '91 Addendum to the Proceedings

[Leathers90] B. Leathers, cited by K.C. Burgess Yakemovic, Workshop: The Bottom Line: Using OOP in a Commercial Environment, OOPSLA '90 Addendum to the Proceedings

[Page89] T. Page, Jr., S. Berson, W. Cheng and R. Muntz, An Object-Oriented Modeling Environment, Proc. OOPSLA '89

[Rubin88] K.Rubin, P. Jones, C. Mitchell and T. Goldstein, A Smalltalk Implementation of an Intelligent Operator's Associate, Proc. OOPSLA '88

[Rubin92] K. Rubin and A. Goldberg, "Object behavioral analysis", CACM, 35 (9), 48-62, 1992

[Rumbaugh87] J. Rumbaugh, Relations as Semantic Constructs in an Object-Oriented Language, Proc. OOPSLA '87

[Rumbaugh91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, Object-Oriented Modeling and Design, Prentice Hall, 1991

[Seidewitz87] E. Seidewitz, Object-Oriented Programming in Smalltalk and Ada, Proc. OOPSLA '87

[Spinelli94] A. Spinelli, P. Salvaneschi, M. Cadei and M. Rocca, MI - An Object Oriented Environment For Integration Of Scientific Applications, Proc. OOPSLA '94

[Tonge97] F. Tonge, private communication, June 1997

[Vayda95] T. Vayda, Lessons From the Battlefield, Proc. OOPSLA '95

[Wirfs-Brock90] R. Wirfs-Brock, B. Wilkerson and L. Wiener, Designing Object-Oriented Software, Prentice Hall, 1990