

Creating a Traceable UML Model

Thomas A. Bullinger
President
ArchSynergy, Ltd.
Victor, New York

Sandeep Mitra
Associate Professor
SUNY Brockport
Brockport, New York

Class ESC-426

Introduction

Since the early 1980's, modeling has been prescribed for helping software developers to understand their systems as they analyze the problem, design a solution, and implement and deliver an application. Modeling techniques are not new to the engineering world, as models have been in use since the early days of building architecture. The recent application of modeling techniques to software development is a perfectly natural extension to the software engineering process.

The dictionary tool on our OS X Macintosh computer defines a model as “a simplified description, esp. a mathematical one, of a system or process, to assist calculations and predictions.” A software model is therefore a simplified representation of the software we strive to create. The software model and the modeling process itself serve several important roles in the software development process:

- Represent the problem the application is intended to solve
- Represent the behavior and structure of the application to be built
- Illustrate the mapping of technology into the desired solution
- Provide a consistent means of capturing and communicating information
- Lend focus to the development effort

The importance of creating a model in a software development effort cannot be over emphasized. However, the creation of a usable model is no easy task. To fulfill the roles as described above, the resulting model must be built to a well-defined standard. The following attributes describe a well-defined model:

- Must be self-consistent
- Must present multiple levels of abstraction / detail
- Must be complete
- Must facilitate communication among the various stakeholders
- Must allow easy changes over time

The challenge of creating a model with these attributes faces every development effort. The following pages discuss one technique for created a UML model that serves the specified roles of a model, and embodies the described attributes.

Creating a Traceable UML Model

Model the Problem

The first step in creating a suitable model is to understand the problem to be solved by your application. Any software development effort begins with identifying the need for an automated solution. The need is almost always derived from an existing problem that is currently accomplished by a manual process, or one that is only partially automated. The software solution seeks to decrease the manual nature of the current workflow, and facilitate the process as automatically as possible.

Once the need is identified, a problem statement is drafted to describe the problem and bound the scope of the desired solution. A problem statement is a brief paragraph or two describing the problem to be solved and any constraints that might apply. The problem statement should be keynote in nature, or suitable for an elevator pitch (a quick discourse you might offer to your fellow passengers on the way to the 10th floor). Figure 1 is an example of a problem statement.

Wegmans (a local grocery food chain) would like an application to track the flow of goods in and out of their supermarkets. Goods are ordered from vendors and delivered to each store. Vendors are paid for their goods. The goods are placed onto shelves accessible to customers. Customers remove the goods from the shelves, pay for them at the checkouts and take them home. New goods are ordered to replace the sold items. Inventory projections are desirable to ensure consistent stocking levels. The price of goods must be set to optimize sales and profit margins.

Figure 1 - The Problem Statement

After defining the problem in a textual manner, the domain of the problem is explored to identify each of the potential stakeholders in your development effort. The application context can be captured in a UML communication diagram (aka: collaboration diagram) that illustrates the application in the center, and arrays each of the stakeholders around the application. The initial set of stakeholders are derived from the description of the problem statement. Additional stakeholders are identified by interviewing already identified stakeholders, as well as domain experts. Each of the stakeholders are connected to the proposed application and the connections are labeled with the information that flows to or from the stakeholder. The information labels are a hint only, as the details of the interactions

Creating a Traceable UML Model

are specified later in the model. Figure 2 illustrates one example of an application context diagram suitable for our problem statement above.

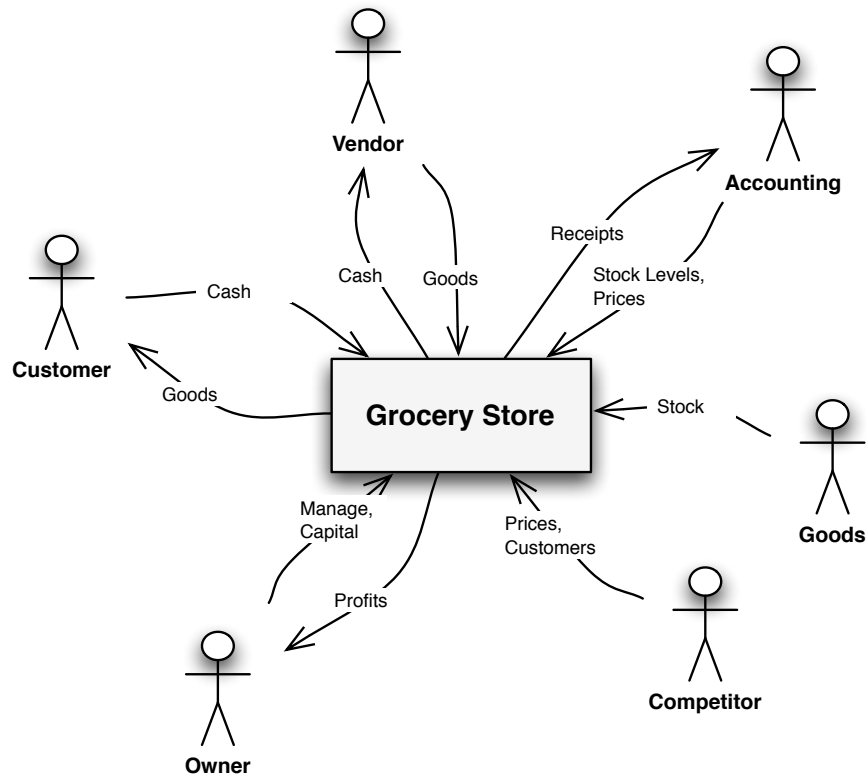


Figure 2 - Application Context

Once the initial stakeholders are identified on the application context diagram, each stakeholder is further described in a table format. The information or requirements provided by each stakeholder are explored, as are the information provided to the stakeholder by the application. Information may only flow in one direction between the stakeholder and the application. In many cases, stakeholders are identified and added to the application context diagram, but are shown interacting only with other stakeholders and not the application. The presence of the stakeholder on the diagram indicates that they have been considered, and found to be unrelated to the application. For the application context diagram in figure 2, there are 6 stakeholders. Figure 3 represents one profile for the Accounting stakeholder.

Creating a Traceable UML Model

Name:	Accounting
Description:	The person or organization the manages the financial affairs of the grocery store
Contribution:	
	<ul style="list-style-type: none">• Stocking levels• Pricing
Benefits:	
	<ul style="list-style-type: none">• Receipts

Figure 3 - Accounting Stakeholder Profile

For each of the stakeholders that have a direct interest in the application, each is interviewed, observed or otherwise explored to determine their requirements with respect to the application. Of primary interest is developing an understanding of the behaviors of the participants in the current workflow, and an understanding of the how the current workflow should be updated to facilitate the automation process. Note that a stakeholder may take the form of an existing system or solution. The influence of the legacy systems must also be considered, especially if they will remain part of the new solution workflow. Each of the behaviors of the stakeholders are captured as user stories.

A user story (as derived from the concept in eXtreme Programming), is one complete flow through the system from start to finish, and accomplishes a discrete goal for the “user” of the system (or more accurately, a role the user may play). User stories are initially captured in diagram form using UML use case notation. The primary difference between a user story and a use case is that a user story takes place, and is described solely in the problem domain. Use cases describe a solution, user stories describe the problem. The collection of user stories are derived from an analysis of the stakeholders and their profiles.

As with any complex model, the user stories may be numerous. To ensure the collection of user stories remains maintainable and easy to read, they are modeled into a hierarchy of relationships. The hierarchy ensures that any one diagram remains readable by limiting the number of elements to 7 +/- 2 [1]. The inclusion of the hierarchy is also consistent with the five attributes of a complex model [2]. The means by which the hierarchy is decomposed is essential to ensuring a traceable UML model.

A hierarchy is created in one of two manners. The natural technique is to select a group of diagram elements which have a common relationship to each other, and refactor (move) them to another diagram. A common relationship between elements can be intuited in several ways. The first technique is to choose items that have a behavioral relationship that is a subset or component of the primary behavior. Known as functional decomposition, this technique

Creating a Traceable UML Model

breaks a single complex behavior into multiple, simple behaviors, each strung together at a higher level of abstraction for the complete behavior. While effective for small scale systems, this technique forces the decomposed behaviors to include the context of the higher-level behaviors, limiting the ability to reuse the sub-behavior elsewhere. Most contemporary software uses functional decomposition for hierarchical decomposition limiting the ability of the application to evolve over time and be reused in meaningful ways.

The second technique for decomposition is to choose items that are related by a unifying concept or theme. Dubbed *Abstractive Decomposition*, this technique reflects the philosophy of inheritance found in object-oriented programming languages. Abstractive decomposition groups a set of elements based not on a common behavior, but rather on a common philosophical abstraction. Hierarchies created in this manner are characterized by the consistency of the sub-elements with the name or concept of the higher abstraction. Abstractive decomposition enables new sub-concepts to be introduced without affecting the higher-order abstractions. Abstractive decomposition is the fundamental relationship found in most design patterns [3].

Figures 4 and 5 represent the user stories for our example problem. Note that the two user story diagrams are related through the use of the subsystem icon in the ellipse for 5. Report Results. The second diagram is labeled to match the ellipse with the subsystem icon exactly to ensure traceability.

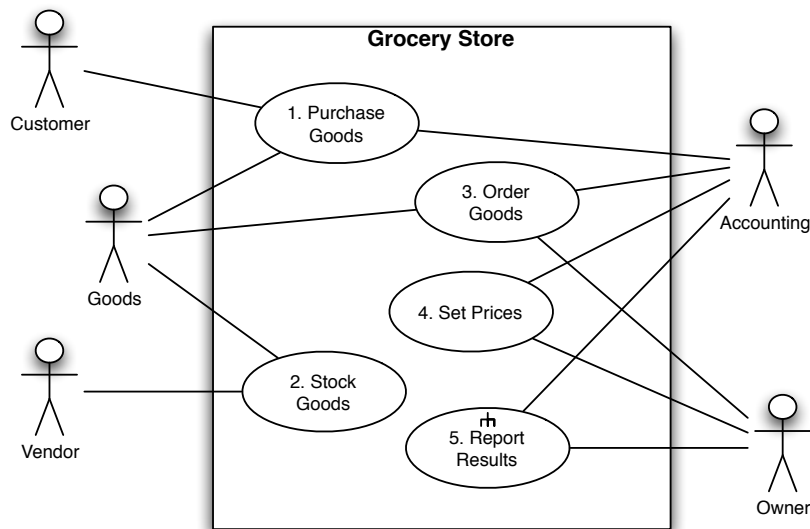


Figure 4 - Grocery Store User Stories

Creating a Traceable UML Model

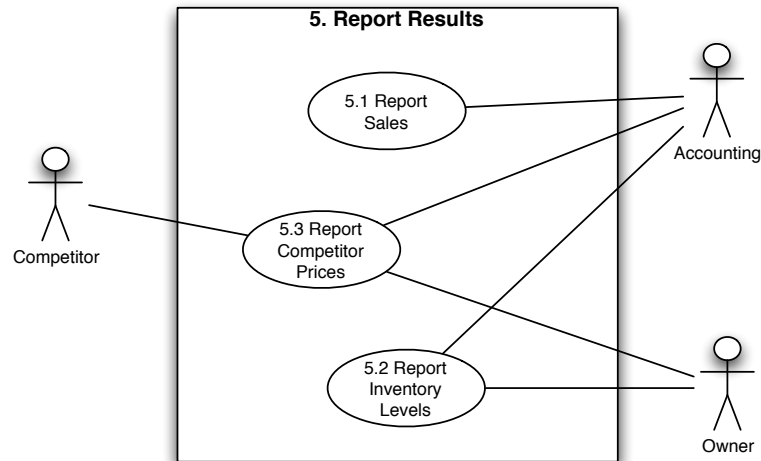


Figure 5 - Report Results

While not always the case, note that each of the stakeholders from the original application context diagram have been considered as actors as each stakeholder has a role to play with respect to the system under analysis. The names of the stakeholders have been carried verbatim into the actors to ensure traceability. The user stories are also prepended with a nested decimal notation to help represent the hierarchy of user story abstractions. The numbers also facilitate traceability throughout the model.

Once the first attempt at user story diagrams is complete, a textual description is created for each of the user stories where applicable. As some of the abstract user stories may be placeholders for the sub-user stories (as in 5. Report Results in figure 4), many of the descriptions will be limited to the “leaf” user stories in the hierarchy. As this is still analysis, the user stories describe only the behaviors in the problem domain, with problem domain entities. The textual descriptions for the user stories include the following information:

- Name - The name of the user story exactly as seen on the user story diagram. The name must match to ensure traceability.
- Description - A brief description of the user story if the name is not sufficient.
- Preconditions - The things that must be true for this user story to proceed.
- Workflow - A sequence of events or script that describes “who does what to whom” for this user story.
- Results - The end, measurable result of the workflow.
- Variants - Alterations in the normal workflow.
- Exceptions - Things that can go wrong, or abnormal alterations in the normal workflow.

Creating a Traceable UML Model

Figure 6 represents the user story for 2. Stock Goods, and is labeled appropriately to exactly match the ellipse as seen on the user story diagram.

Name:	2. Stock Goods
Description:	A vendor delivers goods which are placed on shelves in the store
Preconditions:	
<ul style="list-style-type: none">• The grocery store has shelves available for goods• The goods have been previously ordered	
Workflow:	
<ol style="list-style-type: none">1. The Vendor delivers goods to the store loading dock2. Accounting pays the Vendor for the delivered goods3. Accounting increments the inventory of the delivered goods by the amount supplied by the Vendor4. Accounting notifies the Stock Person of the arrival of the goods5. The Stock Person gets the goods from the loading dock6. The Stock Person determines the designated location of the goods from inventory7. The Stock Person transports the goods from the loading dock to the designated shelf	
Results:	
<ul style="list-style-type: none">• The goods are available on shelves for Customers to purchase• The inventory of available goods is increased by the amount provided by the Vendor	
Variants:	
<ul style="list-style-type: none">• The Vendor provides goods without a purchase order	
Exceptions:	
<ul style="list-style-type: none">• If the goods provided by the Vendor are not listed in the inventory, they are added	

Figure 6 - User Story Description

At the completion of the user stories, the behaviors as observed in the problem domain are captured in the diagrams and text descriptions. The user stories are the most difficult of the modeling process, as they require the most interpretation of information. Once the user stories are identified and described, the remainder of the model flows naturally from this information.

The user stories provide a suitable framework for understanding the behaviors of the desired application, but are not sufficient to completely describe the problem. To ensure that we understand the assignment of roles and responsibilities to the various participants in the workflow, a UML sequence diagram is essential. The UML sequence diagram is based on the *nouns* in the user story description, and illustrates the dynamic interaction between the participants in the user stories. The sequence diagram shows how the various behaviors are assigned to each participant. The sequence diagrams are derived from the user story

Creating a Traceable UML Model

descriptions by analyzing the workflow in the description of each user story, and mapping the workflow onto the activities represented by the sequence diagram. If the activities in the user story workflow are in the form of “who does what to whom”, the translation to the sequence diagram is fairly straightforward. If not, then some thought must be applied to interpret how the behaviors are mapped. Each user story will have at least one sequence diagram, and the sequence diagram will be labeled with the exact name of the user story. If the user story has significant variants or exceptions that alter the workflow, additional sequence diagrams may be created to represent them. Figure 7 represents the sequence diagram derived from the user story seen in Figure 6 above. Note that the events transcribed into the sequence diagram match the steps in the workflow of the user story description.

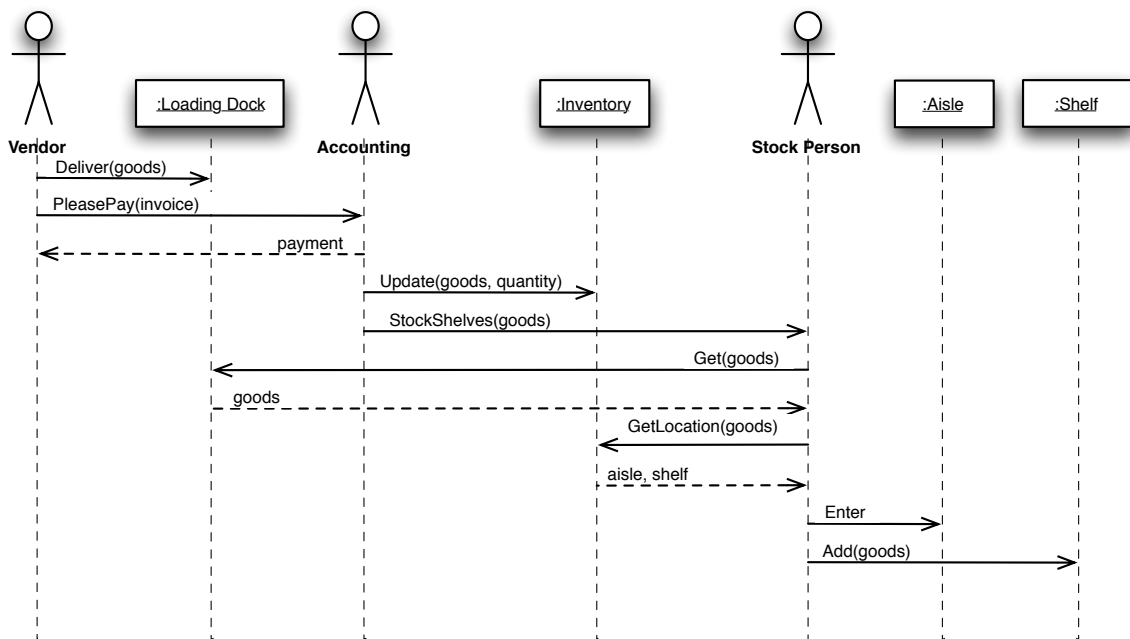


Figure 7 - Stock Goods Sequence Diagram

Sequence diagrams are excellent at representing a complete flow through the system from start to finish, and are therefore well suited to represent most user stories. However, there are many behaviors found in the real world that include branches or decision points, or multiple, parallel behaviors. User stories may also include a large number of variants or exceptions. If your user story fits this description, a UML activity diagram may be better suited to represent your behaviors.

The UML activity diagram is based on *verbs* in the user story description, and illustrate the flow of control between the various activities. The events described in the user story workflow are each mapped to activities on the diagram, and the flows between activities represent the branches of control based on loops, decisions, and other events. As with the

Creating a Traceable UML Model

sequence diagram, the activity diagram is labeled with the exact name of the user story to ensure traceability back to that source. The primary limitation of an activity diagram is that it does not readily illustrate the mapping of the activities to the participants in the workflow. Consequently, it is more difficult to map an activity diagram into an appropriate object-oriented model. To mitigate this limitation, we suggest the use of partitions (aka swim lanes) in the activity diagram to map the activities onto the participants. Each participant will have their own partition, and their assigned behaviors are contained within their respective partitions. The flows between the activities clearly illustrate the dynamic interaction between the participants. In most cases, a user story will map into either a sequence diagram or an activity diagram depending on the nature of the workflow description.

Figure 8 represents a simple activity diagram for the user story 3. Order Goods.

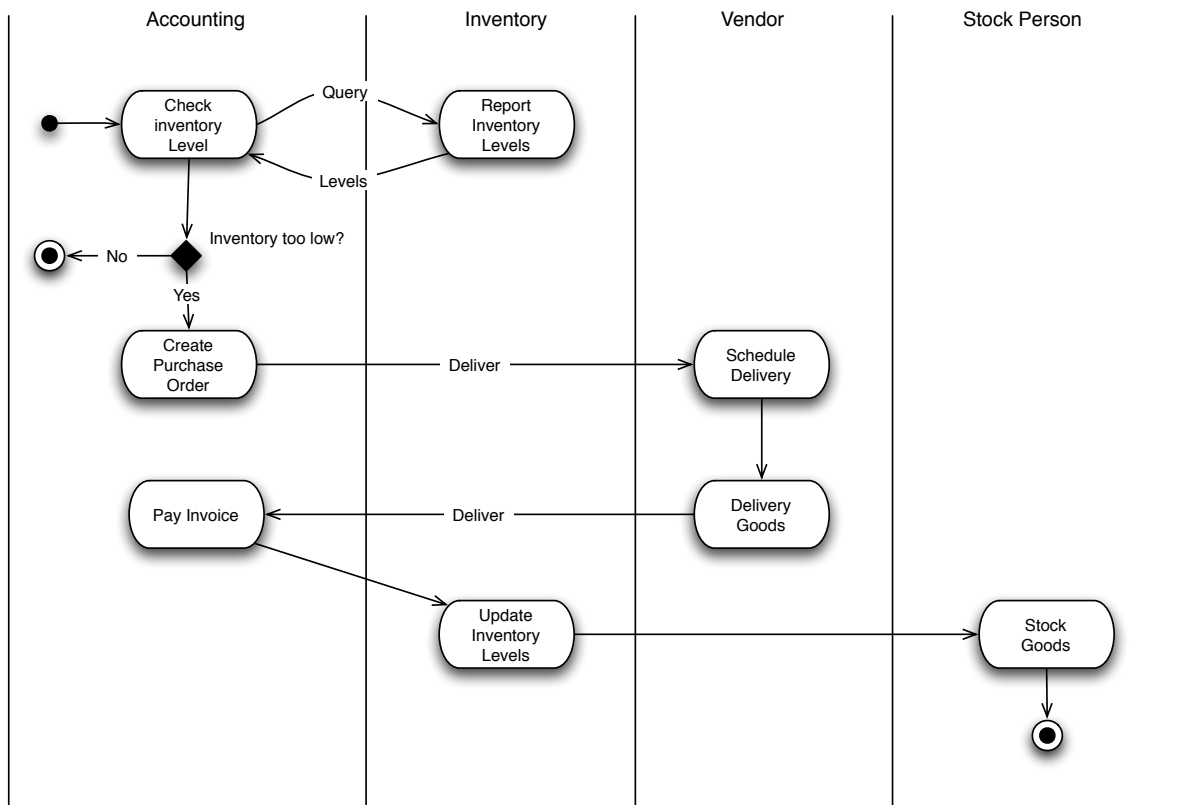


Figure 8 - Order Goods Activity Diagram

As the sequence and activity diagrams are under development, it may become clear that certain participants in the model have behaviors that change depending on the temporal sequencing of the interactions. If this is the case, the various states for that participant can be readily illustrating using a UML state diagram. The state diagram shows the various states the participant can be in, and the available behaviors within each state. The states and events that

Creating a Traceable UML Model

trigger the state transitions must be consistent with the behaviors described in the sequence and activity diagrams in which the entity participates. In every case, the state diagram is limited in scope to a single participant (or perhaps the entire system). The state diagram carries the name of the participant whose states it represents to ensure traceability. Figure 9 illustrates a simple state diagram for the participant Goods.

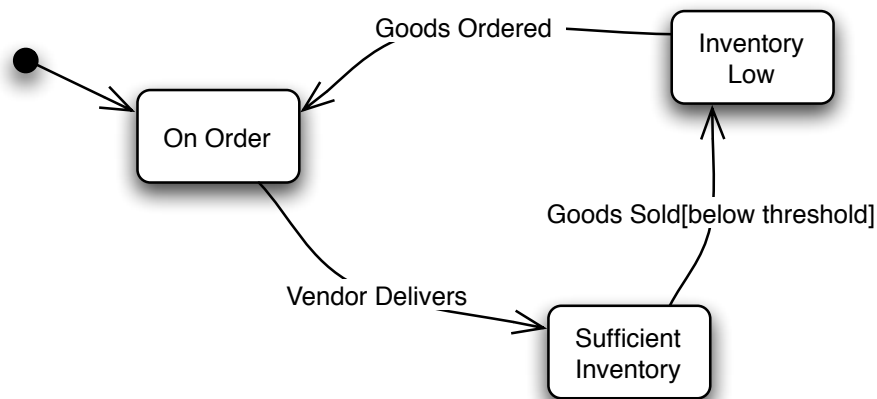


Figure 9 - Goods State Diagram

As the modeling of the dynamic behaviors progress, and the sequence, activity and state diagrams are created, the static relationships between the participants in the diagrams may become apparent. Static relationships are those relationships that are true both within the context of the running application, as well as outside of it. The static relationships between the entities are captured in simple UML class diagrams. For our purposes, only the simplest of UML notation will be used to represent the static relationships between our participants, and the relationships will be limited to simple inheritance and aggregation. The inheritance relationship shows the hierarchy of classification between participants (or abstractive decomposition of participants). The aggregation relationship illustrates a container-ship relationship between participants. In the case of aggregation, the cardinality of the relationship is included to clarify the relationship. In either case, only the simplest UML notation will be applied, as we don't know many of the details of the relationships yet, and any additional notation may be misleading.

The static relationship diagram showing the participants is a derived artifact. Each of the relationships are seen directly during the creation of the preceding dynamic diagrams, or intuited from the interactions observed in the problem domain. Figure 10 illustrates the static aggregation relationship between our participants.

Creating a Traceable UML Model

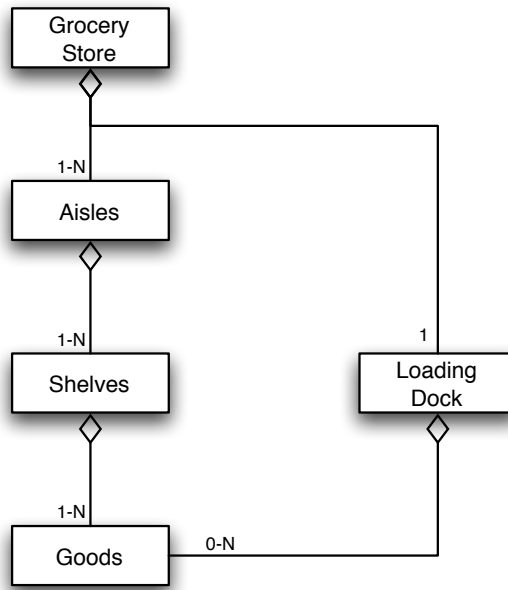


Figure 10 - Aggregation Static Relationship Diagram

As the modeling activities in the problem analysis portion of the project are wrapping up, the roles and responsibilities of each participant remain scattered across multiple diagrams. To clarify each participants' roles and responsibilities, a Participant-Responsibility-Collaboration (PRC) card is created for each participant. Based on a traditional CRC card [4], the PRC card captures the total of the knowledge and behavior for each participant as described across each of the model drawings. By examining the sequence diagrams, each behavior of the participant is seen by arrows arriving at that participant's lifeline. Self-transitions are also included as behaviors. Collaborations are noted by the outgoing arrows triggered by the incoming arrows. Incoming arrows are captured on the PRC card as behaviors, and outgoing arrows as collaborations for that behavior. Analyzing the behaviors of the participant will reveal the knowledge that a participant must maintain to accomplish that behavior. The knowledge is also captured on the PRC card.

For activity diagrams, the partitions in the diagram show the assignment of the activities to the participants. Each activity is pulled off the diagram and captured on the appropriate PRC card. As with the static relationship diagram, the PRC card is a purely derived artifact. All information on the PRC card must exist on the preceding diagrams. It should be clear at this point that the names of each participant are crucial, and must be used consistently across the model to ensure traceability.

Creating a Traceable UML Model

Figure 11 is the Participant-Responsibility-Collaboration card for the Accounting participant.

Type:	Accounting	
Description:		
Responsible for the financial aspects of the grocery store		
State:	Collaboration:	
Know 1 Inventory	Inventory	
Know 1-N Vendors	Inventory, Vendor	
Know 0-N Goods	Inventory, Goods	
Behavior:	Collaboration:	
Pay invoices	Vendor	
Stock shelves	Stock Person	
Update inventory levels	Inventory	
Determine inventory levels	Inventory	
Create and send purchase orders	Inventory, Vendor	
Set prices on goods	Sales Report, Price Report, Inventory	
Create Sales Reports	Sales Report, Inventory, Owner	
Create Price Reports	Price Reports, Competitor, Owner	
Create Inventory Reports	Inventory Report, Inventory, Owner	

Figure 11 - Accounting PRC Card

Manage Change

Up to this point, we've been deliberate about modeling only the problem to be solved, and have avoided the introduction of solution domain constructs. The preceding analysis of the problem ensures that we fully understand the problem we hope to solve. While the problem analysis may seem redundant to those who "already understand" the problem, it is essential to ensuring the traceability of the solution back to the problem. Besides, once the problem is modeled for a particular business, it never has to be done again. The problems addressed by a particular line of business evolve slowly over time. In contrast, the technology applied to solving those problems changes very rapidly. With a solid model of the problem, it is vastly easier to remap the problem into a different solution than to redesign another solution from scratch.

The other critical application of the problem model is an understanding of how changes affect the eventual solution. Without an existing problem model, changes in the problem domain are applied in a potentially arbitrary manner to the solution. With a problem model, changes in the problem domain are easily captured and understood in the context of the

Creating a Traceable UML Model

problem model. With a fully traceable problem analysis model, changes in the problem domain are quickly cascaded through the model, and the impact of the change assessed. With appropriate mapping of an existing problem model into a solution design, the changes identified in the problem domain are easily mapped into the solution domain, and their impact on the existing solution is easily assessed. Without traceability, the impact of change cannot be readily assessed precluding intelligent decisions with respect to content and schedule.

Model the Solution

The model of the problem provides an easy starting point for creating a solution design. The first consequence of our problem model is that we've already divided the system into manageable chunks for starting our design. In fact, if we've been deliberate about sticking to the problem domain for identifying our participants, their behaviors and relationships, then we've already defined the high-level architecture of our solution. All good architectures are firmly rooted in the problem domain, and our problem analysis allows us to root out and identify the appropriate architectural components. The problem model specifies the various components of our system, assigns roles and responsibilities to those components, and defines the relationships between them. The goal of the solution is to correctly implement the identified components.

Each PRC card from the problem analysis defines a component or subsystem for our solution. The design process begins by separating each PRC card, and examining each as a stand-alone component of the solution. The first step in the design process extracts the behaviors from the PRC card and maps them into use cases. Figure 12 is the use case diagram for the Accounting participant described above.

Creating a Traceable UML Model

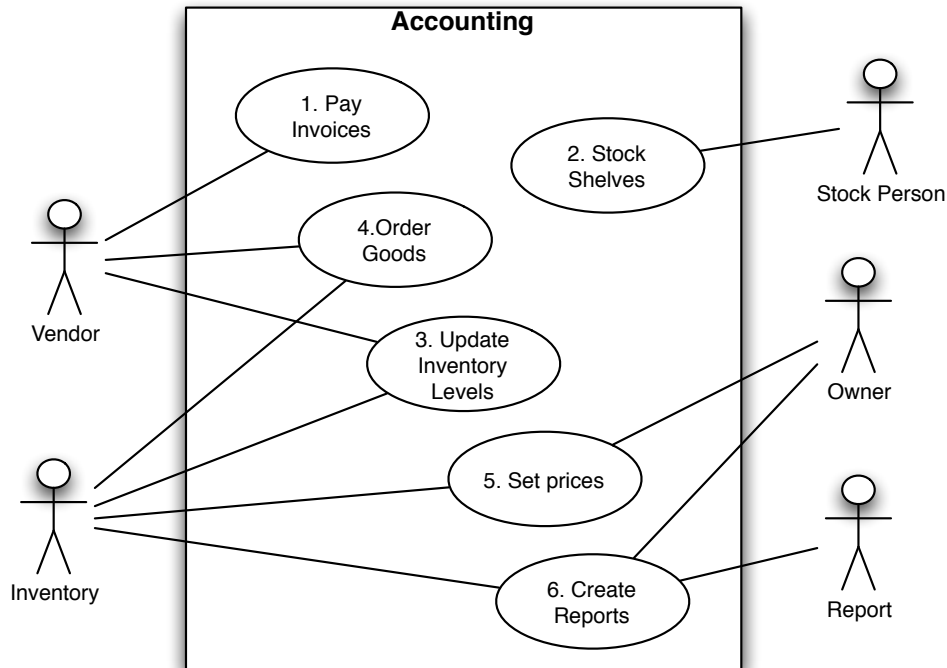


Figure 12 - Accounting Use Cases Diagram

Using UML use case diagram notation, each of the required behaviors are captured in a use case model. Once diagramed, each use case is then described in an identical fashion to the user stories in the problem domain. In the solution domain, there may be additional notation added to the diagrams as the behaviors are more likely to be related. The use case descriptions now include solution domain constructs in the activities that comprise the workflow. At this point, the designer can begin to include databases, protocols, containers, and other solution entities.

Using the same process as described in the problem analysis, the use case workflow is mapped into a set of sequence diagrams and activity diagrams as appropriate. Again, more details will be added to these diagrams to capture the solution domain constructs and constraints. UML state diagrams are created as needed for the solution domain participants to capture the state-dependent behaviors. From these diagrams, the relationships between the classes are captured on UML class diagrams. Throughout the design model, the names of the use cases, and the labels on the diagrams are maintained to ensure traceability between the artifacts.

As before the behaviors of each of the classes are extracted from the sequence diagrams, activity diagrams and state diagrams. The relationships are extracted from the class diagrams. The information is captured on CRC cards as a definition for each class in preparation for

Creating a Traceable UML Model

implementation. The detailed CRC card, when completed, is mapped directly into source code for the implementation. The other UML diagrams created during the solution design phase provide supplemental information for the implementation activities.

We've found that the solution design process requires at least two iterations. The first iteration is a high-level design, and identifies the implementation subsystems, their behaviors are relationships. The second iteration brings in the details required for mapping into an implementation, and defines the classes required for the implementation.

Once the implementation has been created, it is possible to lay out the entire model, both analysis and design, and "connect the dots" between the original observations in the problem domain, and the implementation code that addresses the requirements of the observation.

A fully derived / traceable model also provides the ideal environment for the creation of test plans and test cases. The user stories and the described workflow provide the "greased path" test cases for customer acceptance. The variants and exceptions identifies fill in the details for a reasonably complete customer acceptance test plan. The use cases and their descriptions in the solution domain provide the ideal starting point for engineering test cases, both for white box and black box perspectives.

Conclusion

By considering the construction of a UML model in terms of two domains (problem and solution), and by tying together the various artifacts by reference names, the resulting model is far better positioned to meet the needs of the stakeholders of a development effort.

Creating a Traceable UML Model

References

- [1] Miller, George A. “The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information”. *The Psychological Review* 1956 (vol. 63, pp. 81-97)
- [2] Booch, Grady. 1994. *Object Oriented Analysis and Design with Applications*. 2 Edition. Addison-Wesley. ISBN 0-8053-5340-2.
- [3] Gamma, Erich, Helm, Richard, Johnson, Ralph, Vlissides, John. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.
- [4] Beck, Kent, Cunningham, Ward. “A Laboratory for Teaching Object-Oriented Thinking” *OOPSLA '89 Conference Proceedings* October 11-16, 1989.