Establishing the Completeness and Correctness of a Domain Object Model

by

Crockett Harris Hopper, B.S.

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

May 2006

Establishing the Completeness and Correctness of a Domain Object Model

Approved by
Supervising Committee:

Dedication

I dedicate this work to my beautiful wife Amy and my beloved son Caleb, without whose enduring patience and support I would never have made it this far. Thank you both for putting up with a husband and father who was so often absent or tired or cranky because of this work. I love you both with all of my heart and I'll do my best to show it!

Acknowledgements

Isaac Newton wrote “If I have seen further it is by standing on the shoulders of giants.” This is an appropriate attitude for any student of science or engineering, for no work of scholarship is created without the influence and assistance of countless others. This work would have never been completed were it not for the contributions of many friends and colleagues who have deliberately or even inadvertently provided insight into this research. I am bound to miss some, but these are a few of the people deserving of more appreciation than I can show:

- Dr. K. Suzanne Barber for her excellent work in requirements engineering and software architectures. She established the foundation that got me started on this path and taught me that the software development process begins and ends with the requirements.
- Dr. Tom Graser for keeping me focused on my topic with his keen observations, his many helpful emails, and the insight shared through very carefully prepared homework assignments that RATIONALE is important enough to be spelled in all caps.
- Mark Mayfield for many hours of fruitful object modeling discussion and for introducing me to the fact that object oriented thinking is not so much about objects as it is about objects collaborating by message passing and late binding.

- Charlie, Angie, and Roman Vaught for giving me a quiet place to study and a comfy place to sleep during those very, very long class weekends.
- Todd Williams and Matt Stanton for some great help on class projects and for putting up with my erratic study habits – especially while I was working on this report.

Abstract

Establishing the Completeness and Correctness of a Domain Object Model

Crockett Harris Hopper, M.S.E.
The University of Texas at Austin, 2006

Supervisor: K. Suzanne Barber

An essential difficulty of creating and refining a domain object model is that it is hard to establish that the model accurately reflects the real world entities it is intended to represent. Additionally, it is common to independently develop the user interface for a system in parallel with the development of the domain model. The result is the creation of two largely-independent models that may be difficult to reconcile before implementation. The completeness of the domain object model may be established through the execution of usage scenarios or traceability to requirements gathering interviews. The correctness of the domain object model may be independently demonstrated via usage scenarios and established internally via model checking. Similarly, the User Interaction Prototype (UIP) may be exercised with usage scenarios to establish that the necessary visual elements are present to satisfy the functional requirements. This paper describes a method by which these two models may be coupled in order to better establish the completeness and correctness of each. The UIP serves to ease communication of the models to a wide array of stakeholders by presenting task-oriented functionality through familiar visual elements. The domain object model serves to encapsulate domain behavior in objects that represent concepts of the domain without regard to particular tasks or presentation styles. The linking of the two models serves to minimize any divergence between them and to improve the feedback cycle with the stakeholders early in development. This linkage is established by defining the UIP operations in terms of the domain model. User acceptance tests may then be written in the language of the domain, and the visual elements of the UIP may be exercised to demonstrate the execution of domain tasks.

Table of Contents

List of Figures

1. INTRODUCTION

1.1 The Argument for a Single Conceptual Model

Alan Kay once displayed a video entitled “A Private Universe” [9] to a joint congressional hearing [10] that illustrated that graduates of Harvard University were unable to correctly explain either the reason the seasons change or the reason the moon goes through its phases. Kay also shared that he informally repeated the experiment several weeks later after a talk at UCLA, and he confirmed that it is not only Harvard that is burdened with students, alumni, and faculty that cannot accurately explain these phenomena. Indeed, at each location about 95% of the respondents answered incorrectly. More compelling than the implications for science education is the fact that when Kay was able to ask some follow-up questions, he learned that each respondent already had the knowledge needed to dispel their false notions; for example, even though each stated confidently that the earth is warmer in the summer because it is closer to the sun, they also recognized that when it is summer in the northern hemisphere it is winter in the southern.

Each of these people held two independent notions about the universe that were completely incompatible, yet they had never been brought to a point in their education where they were forced to confront that paradox. Kay attributes this to the way these people had learned science “as isolated cases, stories that would be retrieved to deal with a similar situation” rather than “a system of inter-related arguments.”

It is quite natural for a person to learn and communicate through stories, and the requirements elicitation process is often structured around the gathering of stories to describe what goals a proposed software system should satisfy. Scenarios and use cases are commonly used because they have been shown to be an effective mechanism for revealing the stakeholder vision for the domain under development [6, 7]. This is important because, as Fred Brooks noted in his essay “No Silver Bullet”, “The hardest single part of building a software system is deciding precisely what to build” and “the most important function that software builders do for their clients is the iterative extraction and refinement of the product requirements” [8].

Anything the requirements engineer can do to gain leverage on this central software engineering problem is helpful. However, “A Private Universe” should illustrate that it is quite possible for a human being to happily hold to conflicting notions, and it must be a central objective of the requirements engineer to draw out these notions and compel the resolution of conflict. Therefore, a state of the art requirements engineering process requires an iterative cycle of acquiring, modeling, and synthesizing these narratives into a unified structure that goes beyond a set of stories; it requires a conceptual ontology that forms the basis for a domain specific language that may be shared by all stakeholders. The Software Engineering Process Activities (SEPA) is one such methodology that drives the refinement of stories (interview sessions) into a collection of models that are refined over time and continually verified against the preceding activities. SEPA specifies the construction of appropriate models based on each interview session. These models are then synthesized across sessions until a model to support the entire system scope is produced [1, 2].

Consider that a single stakeholder may participate in dozens of interview sessions over the course of weeks or months on a large software project; on a given day that stakeholder may be participating in a workshop focusing on scenarios, use cases, user interface screens, business processes, or business object modeling. If the requirements engineer fails to refine the narratives and models gathered in each session into a unified whole, the risk of disaster for the project escalates. If the stand-alone session artifacts are not merged into a unified model, then the task of merging the entities and relations discovered in the requirements gathering activities is necessarily deferred until later in the development effort. It is likely that some conflicts arising from the "Private Universes" of the stakeholders will lurk in the system until implementation or deployment; since the set of requirements and models will ultimately yield a single software system it is essential that they be unified as early in the development cycle as possible; if the discovery of these defects is deferred until implementation, the cost of repair may be 50 to 200 times higher than if they are resolved early [16].

The remainder of this section deals with some essential concepts of modeling the requirements of a software system, including exactly what model means in this context and how we may establish that a model is good with respect to the key qualities of consistency, completeness, and correctness. Then in Section 2 we present a detailed case study of a loan application domain where two significant requirements models, a User Interaction Prototype and a Domain Object Model, were not reconciled with one another to the detriment of the project. Section 3 introduces our proposed solution to the conflicting artifacts problem. Section 4 presents the conclusions we reached based on the work presented in this report. Finally, in Section 5 we propose two areas of further research that hold promise for further advances in the requirements engineering discipline.

1.2 What is a model?

Model may be the single most overloaded term in all of software engineering. In a given context the term may refer to a simple box-and-line diagram, a collection of text, or a mathematically formal specification. Michael Jackson proposes that some of the confusion arises from the mixing of two distinct types of models – analytic and analogic. An analytic model refers to a formal specification, such as a finite state machine, that describes the behavior of a system in a realistic fashion; an analogic model refers to an alternate reality that shares some properties with the world [15]. A good example of an analogic model is a road map. It is scaled down and omits a great deal of data about terrain in order to present key information about the highway system in a useful manner. The exclusion of concepts is as important to the quality of an analogic model as those concepts that it retains. It is important to note as well that even though analogic models differ from the world in key aspects, the concepts that are represented have definite properties and relationships that allow for rigorous analysis based on the model.

In constructing software systems we use both the analytic and the analogic class of model. Each type is suited to particular classes of problems. For example, if we need a clear understanding of the branching logic of an elaborate set of business rules, a finite state machine might serve well to illustrate the myriad states that may exist in the system. If instead we decide to construct an object model in order to identify the users of the system and the roles they play, then we are constructing an analogic model; this model ignores countless irrelevant details about the people and systems that are actors in the domain in order to highlight the business related attributes and responsibilities they do or do not share. Such a model is a simplification of reality; its value arises from a careful selection of essential attributes from among many that may be extraneous to a given problem at hand.

Larman makes an excellent point about models right on the cover of his book *Applying UML and Patterns*. There is an illustration of two diagrams; the first is a photograph of a sailboat with the caption "This is not a sailboat"; the second is a UML class diagram containing the classes sailboat, mast, and hull and the same caption "This is not a sailboat." Of course neither the photograph nor the UML diagram is an actual sailboat. The photograph is able to tell us a great deal about the spatial qualities of a particular sailboat; the UML diagram illustrates that some class of boat called sailboat will contain 1-3 masts and 1-2 hulls [19]. These are each analogic models. The photograph is a mapping from three-dimensional space to a scaled two-dimensional image; the class diagram is an extraction of a few concepts in the domain of sailboats into a simple drawing that illustrates some relationships among them. They are each simplifications of the real thing.

1.2.1 Conflicting Model Definitions

The confusion over what it means to model a software system is highlighted by the following list of definitions taken from several different methodologies:

- The Object Management Group's (OMG) Model Driven Architecture (MDA) [22] defines model as "a formal specification of the structure, function, and/or behavior of an application or system."

- The Unified Software Development Process (USDP) and its corresponding Rational Unified Process (RUP) [21] implementation defines model to be "a complete description of a system from a particular perspective."
- Eric Evans' Domain Driven Design methodology [11] defines model as "a system of abstractions that describes selected aspects of a domain and can be used to solve problems related to that domain."
- Wirfs-Brock [20] defines Mental Model as "a simplified view of how something works so you can interact with it."

1.2.2 A Good Model Definition For Requirements Engineering

The Evans and Wirfs-Brock model definitions above are more useful for requirements engineering than the first two, for while the terms 'formal' and 'complete' do not preclude the use of analogic models, they certainly imply the inclusion of the analytic variety. An analytic model is effectively a simulation of some part of the system; for a software system, that simulation may be a prototype or even an implementation.

Yet UML is not as effective a language for expressing this kind of behavior as many implementation languages are; it is likely preferable to construct analytic models in a language that is intended for simulation as well as communication. It is this kind of insight that has led, for example, to the development of workflow engines that are able to execute business process models that can begin as analysis artifacts and be refined into production quality construction artifacts. As Evans notes [11], "a UML diagram cannot convey two of the most important aspects of a model: the meaning of the concepts it represents, and what the objects are meant to do"; and also, "if you do use some technology that allows executable programs to be written in a UML-like diagramming language, then the UML diagram is reduced to merely another way to view the program itself, and the very meaning of ‘model’ is lost. If you use UML as your implementation language, you will still need other means of communicating the uncluttered model.”

So Evans’ definition is a good one for the requirements engineer, since the analogic model is good for describing “selected aspects of a domain and can be used to solve problems” in a given domain. This notion of model is a distillation of deep knowledge of the problem domain under consideration. It allows for careful reasoning about the domain concepts in the form of logically structured text, graphs (with the caveat from above that graphing alone will be insufficient), or simulations.

The essence of an analogic model is that it is different than the real world. The expression of the model may be as formal or informal as the scope of the problem domain requires. For large development efforts it is likely that simulating the model (or at least parts of the model) will pay large dividends in improved understanding of the domain and in the ability of the analysts to interrogate the model and demonstrate that it captures the requirements.

A second good example of an analogic model comes from the air force command centers of World War II [15]. These commanders had two-dimensional tables with tokens representing each plane in the battle. The token planes were placed on the grid in relationship to their positions in the real world, and each time a plane was destroyed its token was removed from the board and returned to a storage bin. As radio communications provided the command center with updated information, the state of the board was updated as well to reflect the current state of the real battle. This analogic model provided helpful information to the commanders about the running air battle, yet it is a two-dimensional representation of an event taking place in three-dimensional space. Its scale is much smaller than that of the real world, and its state is updated relatively infrequently. This model is actually a simulation of the actual air battle.

Modern tactical displays embody the same principles of reducing the information to the essential and displaying only key information to the decision makers. In much the same way that radio signals in the older model drove men to move tokens on a board, a computer simulation of a real world event is updated automatically by telemetry from remote sensors.

The fact that the model is not strictly adherent to the real world does not diminish our capacity to apply rigorous analysis techniques to it in order to answer critical questions about the domain. Of all of the system development artifacts, a good model of the domain is what most encapsulates the principles and practices of the business. While those aspects of the model dealing with technologies like presentation, data management, and system interaction are important to a successful implementation, it is the business object model that most likely captures the trade secrets of an organization.

1.3 Establishing Consistency, Completeness, and Correctness

The meanings of the three terms *consistency*, *completeness*, and *correctness* are central to the verification and validation activities in software development. Where the requirements engineering effort is concerned they can be quite difficult to quantify. *Consistency* is the most straightforward to comprehend as it is simply asserts that all of the concepts that are present in the requirements modeling artifacts have a single meaning and do not conflict with one another [23].

Completeness is much broader and more ambiguous with respect to the requirements because it asserts that every requirement of the stakeholders is captured and represented in the requirements artifacts. For any non-trivial system we run into the "Undiscovered Ruins Syndrome" where infinitely possible degrees of elaboration and refinement of the requirements lead to the realization that "*the more that are found, the more you know remain*" [7]. The IEEE defines a set of requirements as complete *"if and only if it describes all significant requirements of concern to the user, including requirements associated with functionality, performance, design constraints, attributes, or external interfaces"* [23]. This is a challenging problem, and it is critical to customer satisfaction and possibly to contractual obligations that the methods of establishing that completeness is achieved be defined with observable outcomes. These methods may include reviews, walkthroughs, prototypes, and acceptance tests, all of which increase the confidence of the stakeholders in the result. None of these can establish proof of completeness, because in reality the complete specification of the functionality of a new system where there was none before is boundless. The cost constraints imposed by the time and money required to build the system will establish the bounds within which the specification of the system will approach completeness.

Finally, *correctness* is the union of the concepts of *consistency* and *completeness*. For a given set of requirements, they are correct if and only if the artifacts are both complete and consistent for some defined scope. Again, verification and validation activities such as reviews and traceability are required to establish the degree to which the requirements model is correct.

Given these concepts, we can proceed to examine how well they are satisfied by the case study in Section 2 and determine how they may be improved upon.

2. A LENDING SYSTEM CASE STUDY

2.1 Project Overview

This section details a loan origination system that is based on an actual financial services project with the objective of enabling customers to complete a small credit application and have the system automatically fulfill it. The model artifacts presented here are greatly simplified, but retain enough of the original domain complexity to illustrate some of the issues that were faced on this project. It illustrates the need for different model artifacts to deal with the varying stakeholder concerns on the project, yet it also highlights the need for separate model artifacts to be reconciled with each other in order to ensure consistency.

One difficulty that arose on this project came from the desire to mock up screens and get signoff on them from the stakeholders before any additional domain analysis was performed. The screens were thus in a fairly hardened state by the time some of the critical domain concepts were well understood. The modeling of the human interface can certainly proceed in a somewhat parallel track to the modeling of the core domain, but it is important to identify the core domain concepts so that a common language can be shared across modeling artifacts.

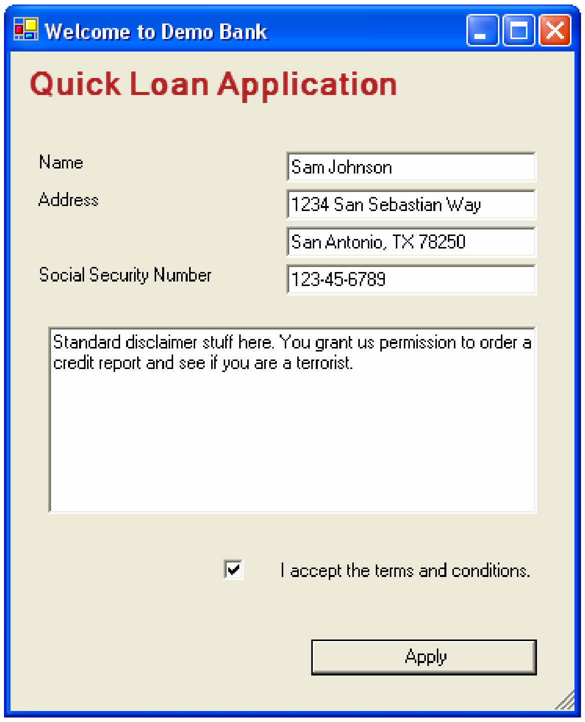
A side effect of the early screens was that the requirements gathering sessions focused on the screen interaction. A set of scenarios was then written in terms of human interface concepts and lacked any reference to core domain concepts because they had not yet been defined. The screens and the resulting scenarios were closely aligned with the current manual process of the business, yet the whole point of this project was to introduce a dramatic change in the business process for fulfilling applications; it was to replace many manual steps with an internet-based application completed by the customer (see Figure 1) and was to be automatically fulfilled by the system in the vast majority of cases.

This oversight led to an assumed one-to-one relationship between users and the roles they play during any given scenario. The screens supposed that a given user would be acting in only one role throughout a session. The result was that there was no facility on an underwriter screen, for example, to perform any fraud analyst operations. In a scenario where the underwriter changes an applicant address, it is possible that this action triggers a request for manual fraud review; since the user is unable to review the fraud item on the same screen, it is likely that the user submits the application with the expectation that it will be immediately fulfilled. In fact, the hidden fraud review request will route the application to yet another queue to be worked by a fraud analyst in a separate user session.

Issues like these are typical for software projects, and in this case the issue was resolved during construction of the system when a user interface developer who was also a domain expert presented the scenario above to various stakeholders. This resulted in a change control to modify the screens and construct a new service that allows underwriters who are also fraud analysts to perform the fraud review before submitting the application. Had the User Interaction Prototype (UIP) been exercised against the domain model during requirements analysis, it is likely that this defect would have been discovered and repaired earlier. By the time the defect was discovered, many analysis and design decisions made earlier in the project constrained the solution to a single screen and a single service modification. This compromise minimized the cost of the change, but it yielded a less elegant solution. Discovering such a requirements defect during the analysis activities allows for much more flexibility in the design of a solution, and it greatly reduces the number of artifacts that are impacted by the change. If we have confidence that the exponential cost curve of defects is accurate, we can easily justify the effort to prototype and reconcile the domain model and the UIP before design is undertaken.

2.2 Key Domain Concepts

This loan origination problem domain has three main classes of users: customers, underwriters, and fraud analysts. Each of these has a need for a different view into the loan application, and each has a custom screen to handle the user interaction. The customer and underwriter screens are shown below in Figure 1 and Figure 2. An object model of the underwriter screen is displayed in Figure 3.


Welcome to Demo Bank

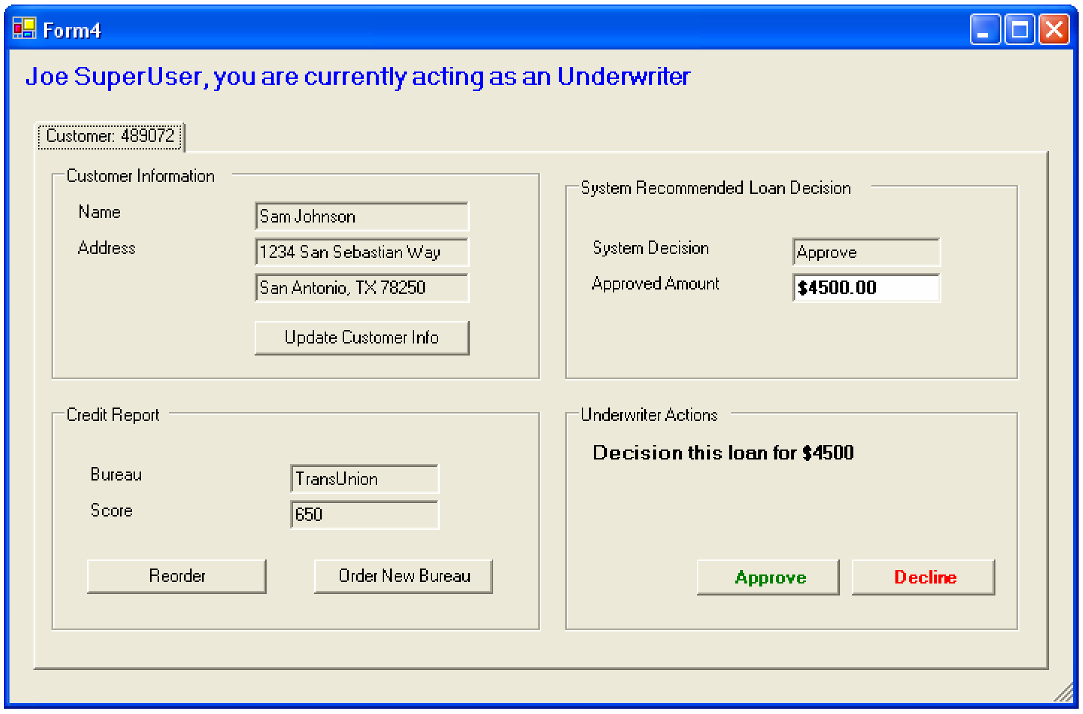
Quick Loan Application

Name	Sam Johnson
Address	1234 San Sebastian Way
	San Antonio, TX 78250
Social Security Number	123-45-6789

Standard disclaimer stuff here. You grant us permission to order a credit report and see if you are a terrorist.

[✓] I accept the terms and conditions.

Figure 1 – Loan Application Customer Submit App Screen


Form4

Joe SuperUser, you are currently acting as an Underwriter

Customer: 489072

Customer Information

Name	Sam Johnson
Address	1234 San Sebastian Way
	San Antonio, TX 78250

System Recommended Loan Decision

System Decision	Approve
Approved Amount	**$4500.00**

Credit Report

Bureau	TransUnion
Score	650

Underwriter Actions

Decision this loan for $4500

Figure 2 - Loan Application Underwriter Screen – Initial

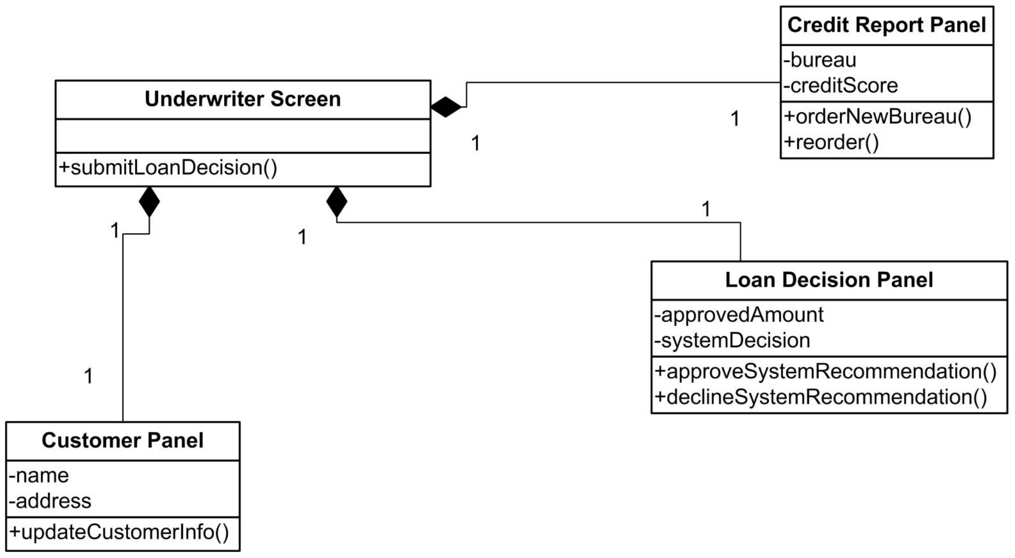
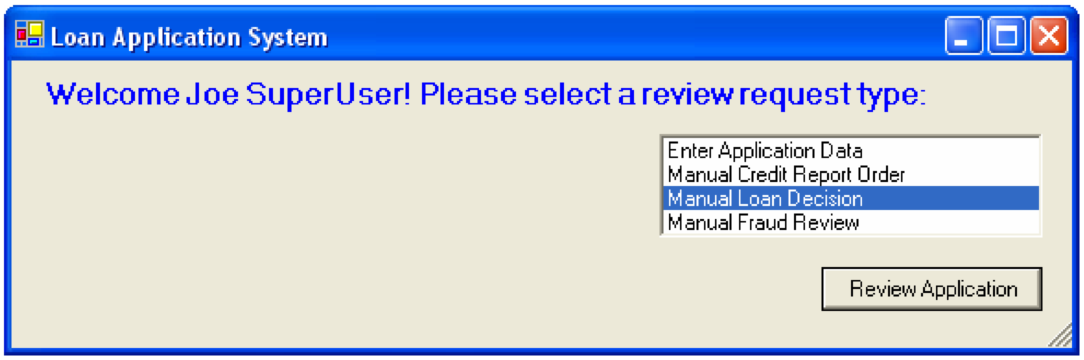


Figure 3 – Loan Application System Human Interaction Model - Initial

A business user requests work from system queues by the type of review they wish to perform. This screen (shown in Figure 4) displays a list of the reviews that the authenticated business user is able to perform. If an application exists that is awaiting the specified type of review, the system displays the appropriate application screen. A user with underwriter permissions, for example, may select ‘Manual Loan Decision’ from the list and have an Underwriter Screen displayed for them to review and decision.



Loan Application System

Welcome Joe SuperUser! Please select a review request type:

Enter Application Data
Manual Credit Report Order
Manual Loan Decision
Manual Fraud Review

Figure 4 – Loan Application Work Queue Screen

The primary business concepts were captured in a domain object model that elaborated on the most salient classes, their responsibilities, and the collaborations between them. This model was often stored as organized text, with some UML diagrams used to highlight relationships between some of the classes. Figure 5 illustrates some of the core classes of the loan application domain.

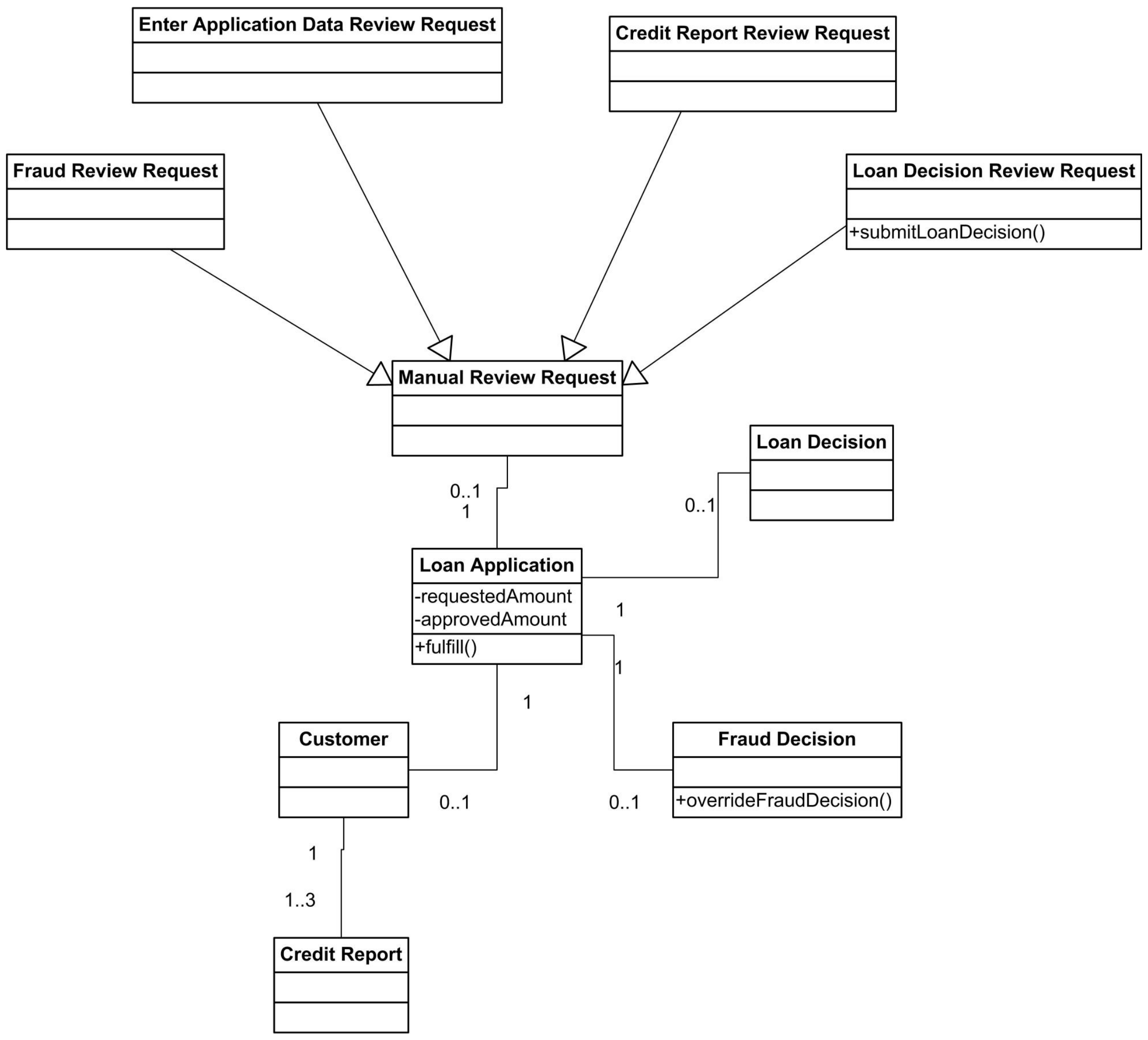


Figure 5 – Loan Application System Domain Object Model

Additionally, the progression of the loan application through its various states from initiation through fulfillment was captured in a business process model. This model was initially represented as a state chart early in the requirements gathering and analysis process in order to capture some complex state transitions that could occur as an application transitioned through its life cycle. Aspects of the model were later refined into a Business Process Execution Language (BPEL) flow during the design process and finally executed by a workflow engine in implementation. Figure 6 is a state chart of the loan application process.

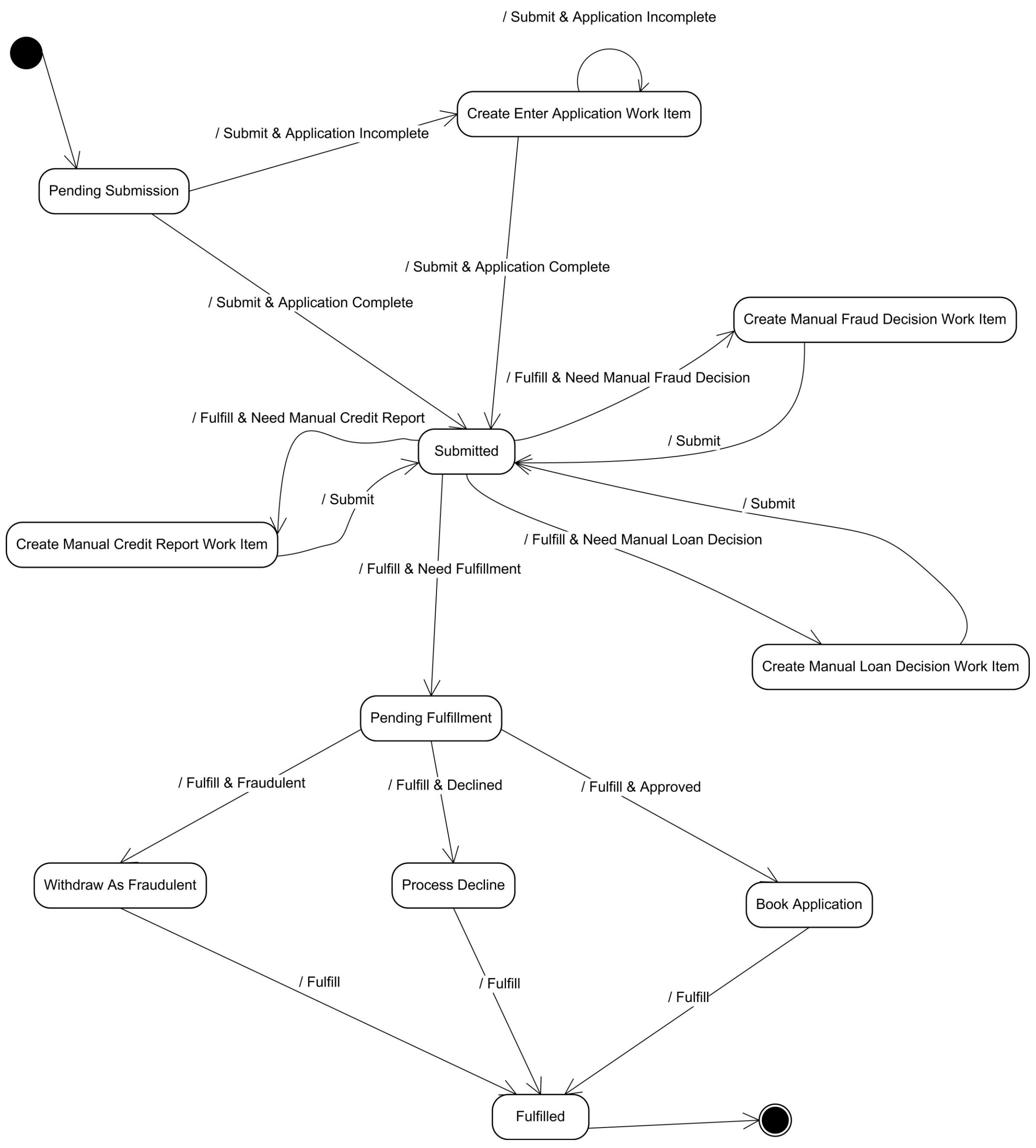

/ Submit & Application Incomplete
Create Enter Application Work Item
/ Submit & Application Incomplete
Pending Submission
/ Submit & Application Complete
/ Submit & Application Complete
Create Manual Fraud Decision Work Item
/ Fulfill & Need Manual Fraud Decision
/ Fulfill & Need Manual Credit Report
Submitted
/ Submit
/ Submit
/ Submit
Create Manual Credit Report Work Item
/ Fulfill & Need Manual Loan Decision
/ Fulfill & Need Fulfillment
Create Manual Loan Decision Work Item
Pending Fulfillment
/ Fulfill & Fraudulent
/ Fulfill & Declined
/ Fulfill & Approved
Withdraw As Fraudulent
Process Decline
Book Application
/ Fulfill
/ Fulfill
/ Fulfill
Fulfilled

Figure 6 – The Lifecycle of a Loan Application.

2.3 System Goals

The primary goal of the system is to automate the fulfillment of loan applications. The main channel through which this process is begun is an internet credit application that a customer may access from any web browser; the customer completes the application and submits it for processing by the bank. The bank's business process is encapsulated by the lending system in such a way that if specified credit criteria are satisfied the system is permitted to automatically book the loan. If any of the criteria for automatic fulfillment are not satisfied, the system will generate a request for manual review and suspend processing the application.

When a request for manual review is generated it is categorized by the type of review that is required; these classes of review are 'enter application data', 'manual credit report', 'manual loan decision', and 'manual fraud decision'. These work request states are illustrated on the business process model (Figure 6). There are two classes of business user identified by the system; these are 'underwriter' and 'fraud analyst'. The underwriter may enter application data, order credit reports, and make loan decisions; the fraud analyst is only authorized to review the fraud requests.

Users pull work from the system based upon the type of review they want to perform. An underwriter may request 'Manual Loan Decision' work from the system and be presented with the information on a loan application that is currently suspended because it requires that its loan decision be made by an underwriter. The underwriter may alternately request 'Enter Application Data' work in order to be presented with an application that was suspended due to a lack of information. Similarly, a fraud analyst may request that the system present a suspended application requiring a ‘Manual Fraud Decision’ before processing may resume.

In each of the manual review scenarios above, once the user has completed their review they submit the application; this returns control to the system and resumes the processing of the application. If all of the criteria are satisfied, the system will book the loan; otherwise it will create a new manual review request and repeat the cycle.

In Section 3 we will examine how the UIP and the domain model artifacts may be improved through a rigorous verification and validation process, particularly when they are validated against one another.

3. RECONCILING THE MODEL ARTIFACTS

3.1 Verification and Validation of the Model

The SEPA methodology asserts that verification is “a study in traceability” requiring that artifacts of the requirements engineering process be traced back to prior artifacts that drove the construction of subsequent versions. This should ultimately enable the identification of the requirement that drove the construction of particular implementation artifacts. This traceability back to the interview sessions with the stakeholders providing the requirements aids in ensuring that the inputs into the models are complete [1].

SEPA also advocates that the models be stepwise refined from each session in order to improve that traceability. On the loan application project a great deal of traceability information was lost because individual interview sessions were not recorded and maintained. The persistent artifacts from the sessions were the screens, UML diagrams, and the business object model, each of which were refined gradually without regard for tracing back to earlier iterations or establishing the rationale for the refinement. This is a key area where SEPA-style traceability would serve to greatly improve confidence in the model.

As construction of the system began and the analysis artifacts of screens, diagrams, and model were consumed by the designers, there was a great deal of confusion regarding what was present in the model, how it was represented, and where in the model a given requirement was captured. While the model was useful to the designers in understanding many intricacies of the problem domain, there was a tremendous learning curve for a designer to become familiar with it and utilize it effectively. In this case traceability forward as well as backward would have served to make the model more approachable for its consumers.

In spite of the traceability shortcomings, the business object model produced in the requirements gathering and analysis activities was extremely thorough in capturing the behaviors of the key domain objects. This appears to be a result of two key factors: the object modelers were experts who had many years of experience in either object modeling or the domain being modeled, and the object modelers applied a catalog of core analysis patterns to the domain that aided in identifying gaps and ensuring completeness.

The analysis pattern technique is cited by Bolloju in a case study of many student teams that improved their object models dramatically after a single, brief training session [17]. The results of this case study are compelling because of the significant improvements observed in the object models of these teams after forty-five minutes of instruction. The same patterns applied in this case study were used on the loan application business object model, and these are the twelve patterns identified in Nicola, et al's Streamlined Object Modeling [5].

The twelve patterns are as follows:

- Actor – Role
- Outer Place – Place
- Item – Specific Item
- Assembly – Part
- Container – Content

- Group – Member
- Role – Transaction
- Place – Transaction
- Composite Transaction – Line Item
- Specific Item – Transaction
- Specific Item – Line Item
- Transaction – Follow-up Transaction

It is especially significant in the context of the loan application domain that one of the patterns, Actor – Role, would have immediately identified the relationship between the underwriter and fraud analyst roles and a single user that could possibly act in multiple roles as they worked with the loan application. This is made clearer with the application of a second pattern, Role – Transaction, that highlights that events that occur on the application are associated with a role rather than a user.

These analysis patterns are generic enough to be useful across many different domains. Indeed, several students from the Bolloju case study remarked that they felt like learning the patterns taught them more about the domains they were modeling [17], even though there is nothing domain specific about the *Streamlined* patterns. That team stated that the patterns gave them "a clear picture of the daily operations of the company." Another team added that they "found it easier to ensure completeness."

These results are made more compelling by the fact that these were novice object modelers. The application of analysis patterns to the problem domain they were attempting to analyze increased the confidence of the teams that their models were complete and correct, and the instructor measured significant improvement in the quality of the models produced. These improvements included the discovery of missed classes, missed associations, the correction of multiplicity in associations, and the renaming of existing associations.

The application of the *Streamlined* patterns and the clear traceability defined by SEPA should be combined to increase the confidence of all stakeholders that the object models have a high degree of completeness and correctness. The activity of tracing model refinements to interview minutes or from interviews to model concepts will reduce the barriers to understanding the model, and the patterns should speed the refinement of the model, making traceability clearer and further promoting the use of the common language established by the model.

Computer Aided Software Engineering (CASE) tools were a popular model driven programming methodology promoted heavily in the 1980's, yet the promise of graphical programming never materialized in the general case and CASE tools were relegated to modeling a few specific domains (such as telecom call processing) that were readily represented using state machines [12]. Today's Model Driven Engineering (MDE) approaches like Lockheed-Martin's Model-Centric Software Development (MCSD) have attempted to incorporate some of the lessons learned from the CASE tool experiments. These focus on developing domain-specific languages for the models and automating the generation of some implementation artifacts from the models [13].

This seems to be a validation of the work of Barber, Evans, Nicola, and others toward a domain centered analysis and design process. The quality of model produced by an MDE tool will be heavily influenced by the quality of the domain language that is developed through the meta-models. These meta-models define the nouns, verbs, and constraints of the model for the domain. The process of defining the meta-model and building other models based upon it is very much the same as growing a domain language using text, boxes and lines, and simulations based upon interview sessions. In either case the objective is the production of a domain model verified by the stakeholders to represent the requirements of the system under consideration.

Once we have achieved a high confidence that the model is an accurate reflection of the stakeholders' intent, that it is verified, the objective becomes to demonstrate that the model is correct with respect to itself and any other requirements and analysis artifacts for the system. In the SEPA process, this validation is often a matter of different levels of review based on a risk analysis; it is also performed naturally as part of the stepwise refinement of session models in the synthesized models that represent the system, as concepts that may have been defined in different ways in different interview sessions are reconciled during synthesis.

Validation is also where the scenarios and use cases that were elicited in the requirements gathering interviews are exercised against the model in order to demonstrate that it is able to respond to them. This exercise greatly aids in traceability because individual requirements may be difficult to map to particular domain concepts. There is not generally a one-to-one mapping of requirements to model entities, but there does tend to be a coupling between requirements and the scenarios and use cases used to illustrate them. Therefore we can trace a requirement to a use case, for example, that will exercise the system in such a way as to demonstrate that the requirement is satisfied. The use cases and scenarios may then be executed against a model simulation, a prototype, or the implementation in order to ensure the requirements are satisfied. This validation may also be performed via walkthrough against a textual or graphical model, however ultimately validation will consist of acceptance tests based upon the scenarios and use cases that are executed against the implemented system [7].

3.2 Improving Quality by Cross-Artifact Verification

Any time we use multiple, independent artifacts to represent a model of the system we introduce the likelihood that the artifacts will diverge into disparate representations of the system. At best that reduces their utility and possibly makes them useless; at worst a conflicting artifact may become a direct cause of serious design flaws in the system. Yet there are also undeniable benefits to using different modeling tools to capture different types of requirements and present the analysis to the stakeholders. Use-cases, scenarios, story boards, screens, and user interaction prototypes are all good mechanisms for representing the types of tasks the users perform in an existing system or will perform in the new one. Relying on these artifacts exclusively, however, leads to a system design that is very much task oriented and will likely be difficult to enhance to support entirely new tasks. In fact, the extreme of use-case driven analysis and design is a functional decomposition of the identified tasks with core domain logic scattered across the screens designed to perform the tasks [18].

In contrast, a domain driven approach as described by Evans, Nicola, and others can produce a rich, analogic model that encapsulates the core principles of a business in such a way that any number of tasks can potentially be written against it with minimal change to the domain model itself. The chief issue with modeling the domain without regard to the user interface is that it becomes difficult to convey the meaning of the model to all of the stakeholders. Even though object oriented thinking is supposed to represent the system to the stakeholders in a domain-centric fashion, many participants in the development effort still have difficulty dealing with objects, messages, and collaborations. The domain model may indeed become a powerful asset to the business for a given project and any number of follow on efforts, but if the essential properties of completeness and correctness cannot be established there will be no confidence on the part of the stakeholders that the model represents their requirements.

One solution to this difficulty is to construct prototypes of the system early in the requirements engineering effort. This implies some sort of user interface, but it may be effective in even a primitive form provided it is able to demonstrate the interaction of the domain objects. Naked Objects is an approach that generates a user interface to allow stakeholders to interact with a running domain model prototype; their DFSA case study reports that the domain centric user interface was so popular with the users that the final implementation continued to present business objects in this fashion [14].

In the loan application case study the user interface was required to conform to a set of standards that were external to the project, so decisions about look-and-feel and technology were set before domain analysis was begun. Clearly in this case a generated user interface was incompatible with the given requirements, but regardless of whether the user interface is generated or constructed, there is no question that we can learn more by exercising the models against each other.

Consider the loan application domain models described in Section 2. Even though the domain object model was produced after the majority of the screens, there was ultimately both a user interaction prototype and a business object model constructed. Yet the absence of early cross-validation of these two artifacts contributed to the following omitted scenario:

A user who can act as both underwriter and fraud analyst is colloquially referred to as a SuperUser.

1. A SuperUser chooses a ‘Manual Loan Decision’ work item from the queue.
2. The application has been determined by the system to be non-fraudulent and the system loan decision is a recommended approval.
3. While working the application the SuperUser changes the customer address to one that is considered high risk for fraud.
4. Underwriter approves the loan and submits the app for processing.
5. The system now routes the application to the fraud queue.

The initial design of the underwriter screen (Figure 2) leaves the user no other option than to submit the loan decision and allow the system to place the application in another manual work queue. This is not what a SuperUser really expects to happen since he also has the ability to take on the role of a fraud analyst. Ideally the system would alert the user to the fact that he just invalidated the old fraud decision and triggered the need for manual fraud review. In this case study however, the UI designers never fully examined a scenario like this.

In the domain model the act of changing an address causes decisions tied to the old address to become inactive (they are retained for audit purposes only). This includes the fraud decision that the system had previously made automatically. Given the new address information, the system flags the application as one that is possibly fraudulent. If this issue is not dealt with before the underwriter submits the loan decision, then the system will generate a request for a manual fraud review. Note in Figure 6 that this review request is not generated until the application returns to the Submitted state from the Manual Loan Decision Review state.

Neither the class diagram (Figure 5) nor the state chart (Figure 6) yield any insight into the fact that the business object for a fraud decision will go inactive when any collaborating objects change. This business rule is embedded in the behavior of the loan application domain object and its collaboration partner FraudDecision. Early in the analysis process some of the business rule behavior was stored only as text, leaving manual walkthrough the only option for validation until a simulation could be constructed.

Regardless, even a manual walkthrough can be a great help to the stakeholders. The text artifact containing the business rule behavior, as noted earlier, could be difficult to navigate. By tracing each of the user interface operations to a corresponding set of business object services we gain a logical entry point into the domain model. As the model and user interface are refined, the walkthroughs can reach as deep into the object collaborations as necessary to illustrate the complete behavior.

The central challenge with this strategy is that it is common for the user interface to be coupled with a controller object layer that manages the task-oriented behavior as a user interacts with the system. The model-view-controller (MVC) pattern is widely prevalent in user interface frameworks and designs today, and it contributes to the flexible behavior of modern interfaces by decoupling the domain logic and the presentation from each other.

Evans describes this additional layer as the Application Layer and defines its responsibilities to include directing the domain objects and possibly storing task-oriented state [11]. Coad does not explicitly identify an additional model layer to map user interface to domain, but he does include sequence diagrams to establish the mapping between the two [4]. This is an easy technique to apply that should work well for manual walkthroughs, and it serves to highlight for the user interface designers precisely what actions are triggered in the system by a user interaction.

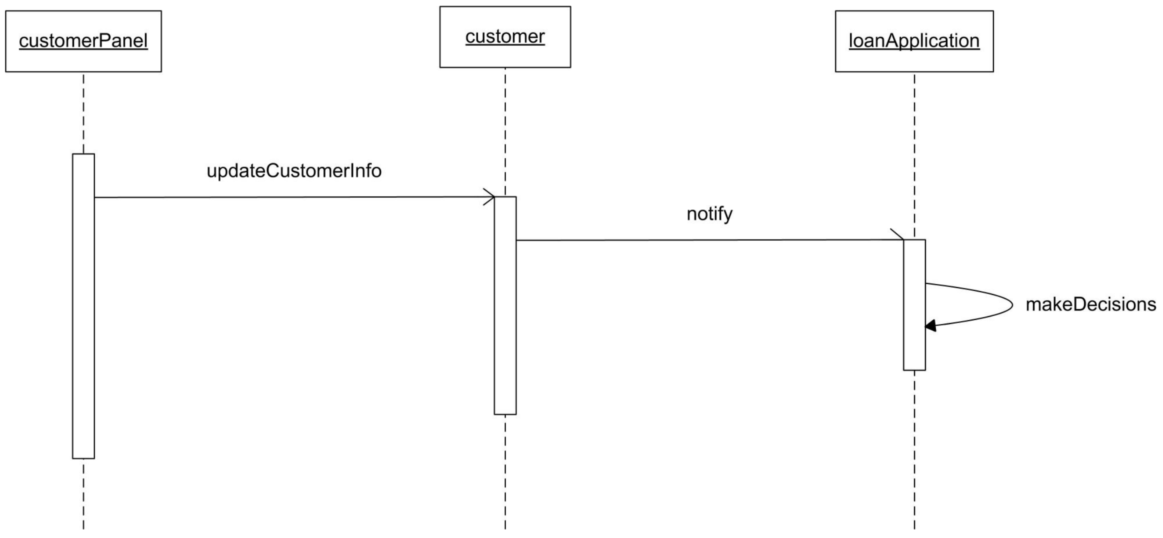
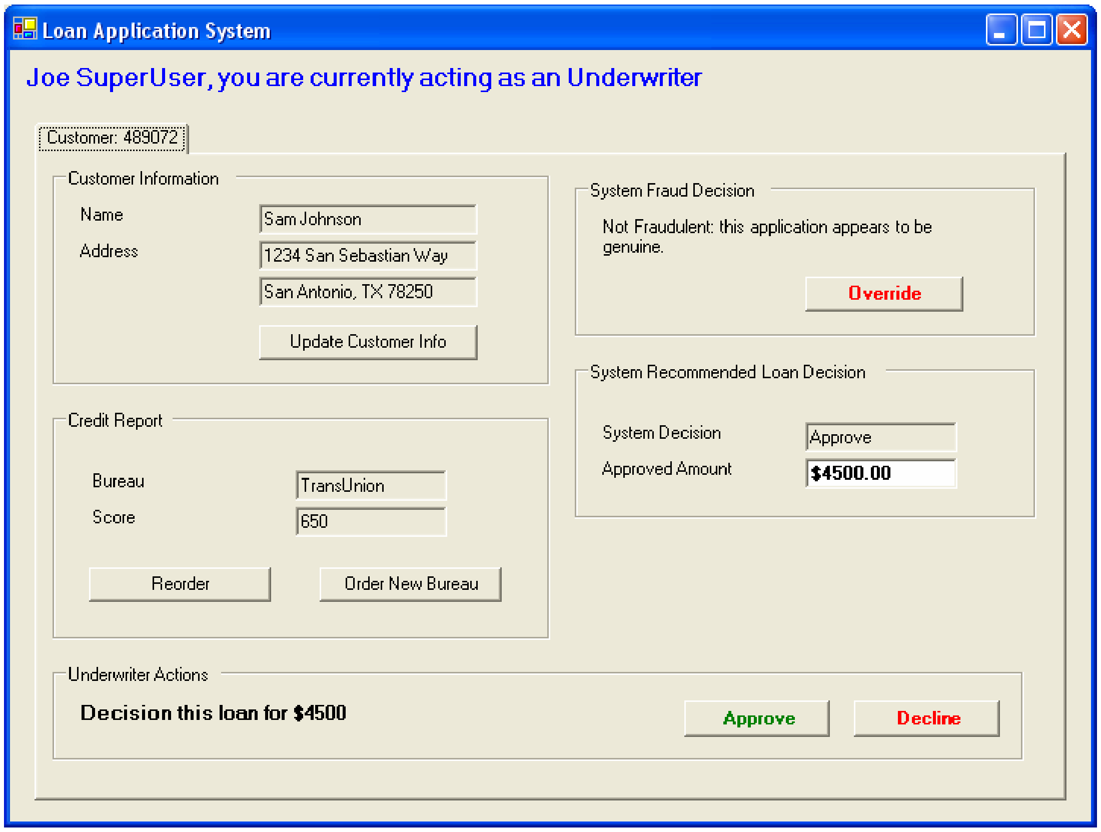


Figure 7 – Update Customer Information Sequence Diagram

In the example loan application, an early walkthrough that illustrated that the act of changing an address could impact the current fraud decision (Figure 7) may have prompted a refinement of the underwriter screen to reflect that additional state information. For an underwriter with no permissions as a fraud analyst, the additional fields may be hidden or read-only; while a SuperUser is provided access to the full power of the business model and can make all of the necessary decisions on a single screen (Figure 8). An updated human interaction class diagram illustrates the additional messages sent when a fraud decision is overridden (Figure 9).


Loan Application System

Joe SuperUser, you are currently acting as an Underwriter

Customer: 489072

Customer Information

Name	Sam Johnson
Address	1234 San Sebastian Way
	San Antonio, TX 78250

System Fraud Decision

Not Fraudulent: this application appears to be genuine.

System Recommended Loan Decision

System Decision	Approve
Approved Amount	$4500.00

Credit Report

Bureau	TransUnion
Score	650

Underwriter Actions

Decision this loan for $4500

Figure 8 - Loan Application Underwriter Screen – Refined

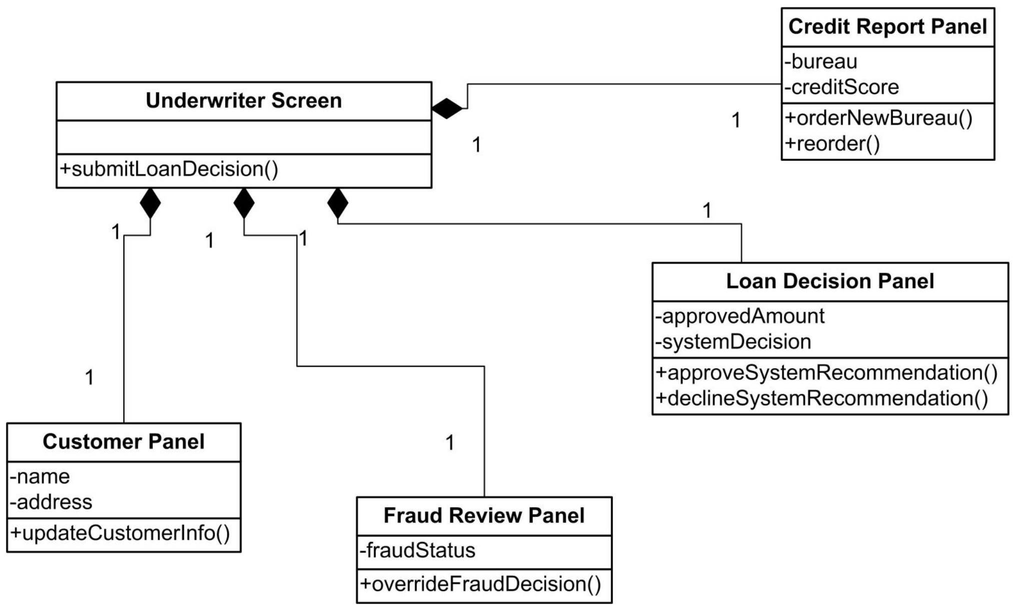


Figure 9 – Loan Application System Human Interaction Model – Refined

Next, Section 4 reiterates our findings in this report and presents our conclusions.

4. CONCLUSIONS

It is likely that the loan application case study would have progressed more effectively had cross-model validation been incorporated from the start. The exposure of the stakeholders to additional views into the problem domain must provide additional insight into the interactions between the user interface and the domain model. In particular, the mapping of user interface operations to domain services plainly illustrates that a single button-click on the interface can result in a number of operations occurring in the application that may have impact beyond what a simple screen displays to the user.

This kind of insight can shorten the feedback cycle between the analysts and the users and increase the probability that an overlooked scenario would be discovered early in the development cycle. If we really believe that the cost of finding defects grows exponentially as it remains in the system, then an aggressive effort in the requirements engineering process to eliminate the vast majority of requirements defects is justified. This must include simulating the model or at least parts of it. The initial cost for building such a simulation is relatively high, but the benefits to analysis are tremendous. The analyst is able to interrogate the simulation and refine it as the requirements are better understood and the model is refined.

This supports the idea of Coad and Evans that a single conceptual model must underlie all of development. While a single text or graphical artifact is too cumbersome to have much utility, a simulation may present any number of views in answer to a given problem. This throwaway prototype may become the embodiment of the 'Ubiquitous

Language' that assists all of the stakeholders in the use of common terminology and meaning as they discuss the system.

As we progress through the development lifecycle, iterating over the analysis, design, and construction activities, it is entirely appropriate that the model be expanded to include business process, human interaction, and deployment model views, among others, but all of these model artifacts must either be transient or synchronized in a dynamic fashion. In support of the single model idea, any artifact representing a view into that model must be generable from it or must be discarded after use; otherwise, the artifacts and the model will inevitably grow out of sync with one another.

Ultimately, the task of reconciling the myriad stories, scenarios, use-cases, and models into a unified whole falls to creative human beings applying their intellects to the problem at hand, but while it is important to remember that fact, it is beneficial to provide tools to aid in the solution. These tools include tracing the model artifacts back to requirements elicitation sessions, ensuring that durable model artifacts are continually synchronized, ensuring that transient model artifacts used to anchor discussions are discarded, and using the different types of model artifacts to improve the validation of each.

One benefit to counter the cost of more elaborate models is that the confidence in the model is increased as feedback is provided to stakeholders and validation is improved by exercising the user interface model against the domain object model. The effort of constructing an executable domain object model simulation may be offset by the elimination of requirements defects very early in the development effort. Section 5 identifies two ideas that merit further research for their promise at reducing the cost of the simulation efforts.

5. FURTHER RESEARCH

While the notion of exercising a domain object model through the use of a custom user interaction prototype seems useful, the cost and complexity of such a strategy may preclude its use in all but the largest development efforts. A worthy alternative to presenting a domain simulation to the stakeholders for evaluation and refinement is the Naked Objects framework. This tool generates interface elements for domain concepts and allows stakeholders to interact with them using a problem solving approach. These user interfaces are unlike the typical task-oriented designs, but the Naked Objects developers report several case studies that indicate users enjoy the power of using the domain objects directly. It would be beneficial to perform additional work with Naked Objects or a similar framework in order to determine if this approach can work within a rigorous requirements engineering methodology.

The student case study by Bolloju using the Streamlined Object Modeling patterns is compelling. After a 45 minute training session these students were able to significantly increase the quality of their models. A second case study using professional analysts and modelers should yield interesting results. SEPA work on the Reference Architecture Representation Environment (RARE) has shown that general modeling expertise may be captured in the form of strategies and corresponding heuristics applied to static properties in order to generate models [3]; it would be useful to see what kind of quality effect the application of these general analysis patterns had on the object models of junior modelers as compared with the decisions made by experts.

REFERENCES

1. K. S. Barber, *Lecture Notes for EE379K*, Requirements Engineering, University of Texas at Austin, spring semester, 2005.

2. K. S. Barber, T. Graser, et al, *Application of the SEPA Methodology and Tool Suite to the National Cancer Institute*, Hawaii International Conference on System Sciences, Maui, HI., 1999.

3. K. S. Barber, T. Graser, *Tool Support for Systematic Class Identification in Object-Oriented Software Architectures*, Proceedings of the 37th International Conference on Technology of Object-Oriented Languages and Systems, 2000.

4. P. Coad, D. North, and M. Mayfield, *Object Models: Strategies, Patterns, and Applications*, Yourdon Press, 1995.

5. J. Nicola, M. Mayfield, and M. Abney, *Streamlined Object Modeling: Patterns, Rules, and Implementation*, Prentice-Hall, 2001.

6. K. McGraw and K. Harbison, *User-Centered Requirements: The Scenario-Based Engineering Process*, Lawrence Erlbaum Associates, 1997.

7. D. Leffingwell and D. Widrig, *Managing Software Requirements: A Unified Approach*, Addison-Wesley Professional, 1999.

8. F. P. Brooks, *The Mythical Man Month, Anniversary Edition*. Addison-Wesley, 1995.

9. *A Private Universe*. http://www.learner.org/resources/series28.html. Harvard-Smithsonian Center for Astrophysics, Science Education Department, Science Media Group, 1987.

10. A. Kay, *written remarks to the Joint Hearing on Educational Technology in the 21st Century, Science Committee and the Economic and Educational Opportunities Committee, U.S. House of Representatives*, October 12, 1995.

11. E. Evans, *Domain Driven Design*, Addison-Wesley Professional, 2003.

12. D. Schmidt, "*Model-Driven Engineering*", IEEE Computer, v.26 n.2, Feb. 2006.

13. D. Waddington, P. Lardieri, "*Model-Centric Software Development*", IEEE Computer, Vol. 26 #2. Feb. 2006.

14. R. Pawson, R. Matthews, *Naked Objects*, John Wiley and Sons, 2002.

15. M. Jackson, *Problem Frames: Analyzing and Structuring Software Development Problems*, Addison-Wesley, 2001.

16. Steve McConnell, "*Software Quality at Top Speed*", Software Development, 1996, http://www.stevemcconnell.com/articles/art04.htm.

17. N. Bolloju, "*Improving the Quality of Business Object Models Using Collaboration Patterns*", Communications of the ACM, v.47 n.7, p.81-86, July 2004

18. D. Firesmith, "*Use Cases: The Pros and Cons*", Knowledge Systems Corporation web site, http://www.ksc.com/articles/usecases.htm.

19. C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, Prentice Hall, Englewood Cliffs, NJ, second edition, 2001.

20. Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener, *Designing Object-Oriented Software*, Prentice Hall, Englewood Cliffs, NJ, 1990.

21. P. Eeles, K. Houston, W. Kozaczynski, *Building J2EE Applications with the Rational Unified Process*, Addison-Wesley Professional, 2002.

22. J. Miller, J. Mukerji, editors, MDA Guide Version 1.0.1, Object Management Group, 2003, http://www.omg.org/cgi-bin/apps/doc?omg/03-06-01.pdf.

23. IEEE Standards Collection, Software Engineering, *IEEE 830-1993*, IEEE, 1994.

VITA

Crockett Harris Hopper, born in Fordyce, Arkansas on December 1, 1971 to Travis and Janis Hopper; he has a long history with computing and software. His first computer was a Timex/Sinclair 1000 given to him by his parents for Christmas when he was eleven years old. He taught himself to program in BASIC by keying in examples found in programming magazines like *Compute!*. He grew up in Hot Springs, Arkansas where he attended Lake Hamilton High School and met his future wife at church when they were both kids. They married under the big Oak tree at Old Union Primitive Baptist Church in June of 1995 where they were both members; as of 1999 they now have a precocious son who enjoys math and may someday soon surpass his dad as the best programmer in the family.

Crockett graduated from Lake Hamilton in 1990 with 14 hours of Advanced Placement credit in English Literature and Chemistry, but his diverse interests made it difficult for him to focus on a single educational goal; he spent the next two years at the University of Arkansas at Fayetteville as an Arkansas Scholar/Byrd Scholar/National Merit Finalist exploring Plant Pathology, Architecture (of buildings, not software), Music Performance (including a stint in the Razorback Band), English Literature, Economics, Food Science, Music Composition, and Music Education. He finally decided to take some time off and work for a while until he realized that if he wanted to actually have family time with his future wife he really needed to get back on track. So in the spring of 1995 Crockett resumed his college education with a focused intensity to get a degree doing the same stuff he had been tinkering with since he was a kid – programming computers!

As an undergraduate in Computer Science/Mathematics at Henderson State University in Arkadelphia, AR, Crockett was exposed for the first time to the theory of computation, networking, and databases. He minored in physics and built gravity simulations as an undergraduate research assistant in the physics department. He was awarded the Bachelor of Science in Computer Science – Mathematics from HSU in 1999. Crockett's professional direction was influenced by a 1997 internship with ALLTEL Information Services where he was able to help design and construct an object repository that was used by other developers to share common objects across projects. Based on that experience and the promise of an advanced software development training program, Crockett went to work for AIS as a client/server programmer in the summer of 1998. That fall he joined the AIS Internet Banking team and had a great time pushing the state of the art in retail banking software by implementing an OFX-based internet interface to core banking systems for several of the top 100 U.S. banks.

In the fall of 2003 Crockett was lured out of Arkansas to USAA in San Antonio, Texas with the promise of more opportunities for pushing the state of the art in software development and an opportunity to attend graduate school at the University of Texas. Crockett does whatever he can to apply software engineering principles to all kinds of interesting problems and continues to try to advance the field of software development.

Permanent address: 11562 Wood Harbor, San Antonio, Texas 78249

This report was typed by Crockett Harris Hopper.