|  |
| --- |
| Fansee INDUSTRIES, inc. |
| Requirements Analysis Modeling Approaches |
| Moving past CMMI level 0 |
|  |
| **Ethan Crawford** |
| **5/7/2007**  **CSS 360 Professor Cioch** |

|  |
| --- |
| An effective Requirements Analysis process is at the heart of every successful large-scale software development project. For a team that is not currently utilizing any requirements modeling techniques, the selection of the correct set of approaches can be intimidating. This whitepaper provides a high-level overview of mature, industry-proven techniques for modeling requirements, using flow-oriented, scenario-based, class-based and behavioral models. This information should be beneficial to any organization wishing to adopt a new requirements analysis process or enhance an existing one. |

# Executive Summary

This whitepaper presents mature, industry-tested techniques for modeling requirements. Four different modeling categories, class-based, scenario-based, flow-oriented, and behavioral, are discussed. Each technique is explained using a real software scenario taken directly from the system that powers Fansee Industries’ flagship product, the Fansee Propeller Beanie. Applied diligently, some combination of these techniques should “make the Fansee Beanie spin!” (see Figure 1).



*Figure 1: artist’s rendering of Fansee Propeller Beanie*

# Introduction – Why model requirements?

There are many reasons to model requirements. Models are an integral part of refining requirements. A rigorous modeling process usually exposes missing scenarios and system components not discovered during requirements elicitation. Models are vital for explaining the system to developers and other interested parties. Just as “a picture is worth a thousand words”, a single well-structured diagram can do the work of paragraphs of text [Pressman05, p. 177].

Models also enable the capture of multiple system dimensions. Not only do they enumerate required functionality, they also help organize functionality into sensible groups, differentiate implementation details from functional requirements, show the flow of data through a system, capture and categorize user experiences, segregate external from internal entities, and establish responsibilities and relationships between objects.

Models also enable many other phases of the software development lifecycle. In the design phase, analysis work artifacts provide a framework upon which design activities build and elaborate. After construction is complete, the same work artifacts provide an excellent basis for validating quality, defined as adherence of implemented features to system requirements.

There are four main approaches to modeling requirements: flow-oriented, class-based, scenario-based, and behavioral. These techniques can be categorized into structural (data and operations are separate) or object-oriented (data and operations are linked) philosophies, although some techniques are applicable to both categories. Each approach has its own emphasis, and the techniques are designed to be used together to properly communicate all relevant system dimensions. Structural models, for example, require at least one behavioral model to show both the “what” (the structure and flow of data) with the “how” (which methods operate on data in a given state).

In this whitepaper, a sample of each technique specific to Fansee Industries is provided. While these samples are not intended to substitute for the complete process of requirements analysis that the team must undergo, they should give the reader some idea of the overall product vision and help provide a business justification for adopting a structured requirements analysis process.

# Flow-oriented modeling

Often considered the “grandfather” of requirements analysis, flow-oriented modeling is concerned with the flow of data within a system. Flow-oriented models are structural, not object-oriented, and they take “an input-process-output view” of the system [Pressman05, p. 194], modeling input vectors that provide raw data, actions that perform data transformations inside the system, and output vectors through which transformed data exits. The primary technique in this category is the data flow diagram.

## Data Flow Diagrams

Data flow diagrams are system-centric, hierarchically-ordered collections of diagrams that utilize a simple library of shapes and arrows to represent “entities” (data inputs or outputs), “data stores” (persistent data storage), and “systems” (some amount of software functionality) [WIKI07a]. The diagram hierarchy is top-down; a level 0 or “context-level” diagram establishes inputs and outputs and represents all the software to be developed as a single product. Figure 2 shows a context-level DFD for the Beanie system.



*Figure 2: Fansee Propeller Beanie System Context (level 0) DFD*

Subsequent diagrams are more detailed in scope. The complete diagram hierarchy provides a “fractal” effect, enabling ‘infinite zoom’ capability into any part of the system. Level 1 diagrams model separate systems within the overall software package; levels 2 and higher show the services within those systems and are grouped by functionally-similar services. A system service is a block of code that enables a portion of the functionality required to implement a user scenario. User scenarios have no bearing on diagram organization or overall system data flow, however. Users are merely alternate input or output vectors and a user’s characteristics influence the individual attributes of data objects. The functional requirements of the system and the data that passes between those interconnected requirements drive the model.

Figure 3 shows a level 1 refinement, with the sub-systems represented as individual circles. The primary inputs and outputs are preserved unchanged, allowing diagram readers to retain context between diagrams and enabling them to incrementally expand their knowledge of the system. Each connecting arrow contains text that describes the action or specific data object being transported or transformed. Finer-grained diagram refinements (not pictured) model the functional requirements of each individual process, showing the flow and transformation of data through the process, with data flowing uninterrupted at every level.



*Figure 3: Fansee Propeller Beanie Level 1 DFD*

DFDs model only data flow and contain no information regarding implementation, class hierarchies, or user experience. Given that most modern systems are developed from an object-oriented perspective and place a heavy emphasis on customer satisfaction, it is unlikely that a development team can be successful relying exclusively on this modeling technique. However, DFD structure is extremely flexible, making the technique ideal as a supplement to other modeling techniques.

# Scenario-based modeling

Scenario-based modeling is oriented around the user. Rather than orienting the model around system functional requirements or underlying architectural structure, scenario-based modeling derives system requirements through enumerating unique interactions of users (“actors”) with the system. This approach is justified by the rationale that the most relevant measure of product success is customer satisfaction [Pressman05, p. 186]. The main techniques used in scenario-based modeling are use-cases, activity diagrams, and swim-lane diagrams.

## Use-Cases

A use-case is a description of a user scenario, written from an actor’s perspective. Use-cases are an integral part of the Rational Unified Process and are a starting point for a large number of other requirements analysis activities. They are constructed by matching a user requirement with an actor and providing a goal, a set of preconditions, a trigger, and a description of the functional steps required to fulfill the user scenario. The generic user interface vector (“channel to actor”) is identified and secondary actors and channels are listed “Exceptions” (error conditions) are listed separately. The case is assigned a set of metadata, which can consist of priority, availability estimate, frequency of use metrics, open issues, and other project-specific info. [Pressman05, pp 190-191]. Figure 4 shows a sample use-case containing these elements.

|  |
| --- |
| **Use-case:** Access location information about friends on a Buddy List  **Primary Actor:** Registered user  **Goal:** Provide a way for an FPB user to physically locate his or her friends  **Preconditions:** Registered user provides correct credentials; user has more than 0 friends added to the Buddy List; user’s friends have not blocked access to their information.  **Trigger:** A registered user decides to see where his or her friends are, at that moment.  **Scenario:**   1. The registered user enters their credentials on the login page 2. The system displays a customized home page for the registered user 3. The registered user clicks the “Buddy List” tab 4. The system displays a table of buddies, a zoomable map, and a set of icons on the map representing each Buddy 5. The registered user selects one of the Buddy icons and clicks “Details” 6. The system displays up-to-date location information for the buddy, including major landmarks, time spent at that location, and previous locations.   **Exceptions:**   1. Invalid credentials when logging into website 2. No Buddies configured 3. No Buddies found within given latitude/longitude parameters 4. Buddy has blocked the registered user from seeing location information or details   **Priority:** Moderate (priority 2)  **When available:** Second iteration  **Frequency of Use:** Frequent  **Channel to actor:** Via PC-based web browser and mobile device application  **Secondary actors:** System administrator, Buddy Beanies  **Channel to secondary actors**: Via PC-based web browser, server-side administrator application, cellular data connection and GPS  **Open issues:**   1. What are the infrastructure requirements to support this scenario? 2. What is the best way to handle latency issues? 3. How much tracking data should be cached, in the event that connection is lost? |

*Figure 4: Sample Fansee Propeller Beanie use-case*

When writing use-cases, it is vital to use consistent nouns and verbs to describe the user scenario. In later phases, this structured grammar will be distilled into classes and data objects.

## Activity Diagrams

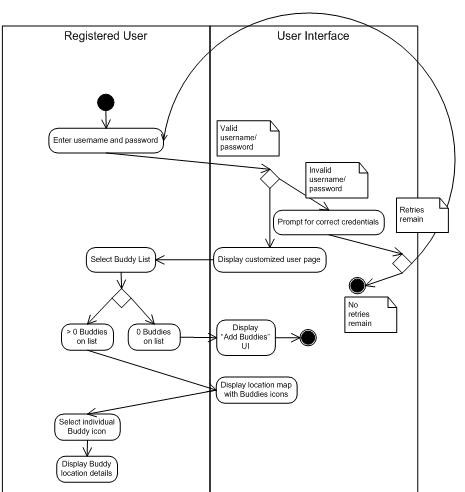
Activity diagrams describe use-cases in graphical form. Preliminary activity diagrams model the entire system, while use-case-level activity diagrams model individual use-cases. The activity diagram hierarchy is much flatter than data flow diagrams (only a few levels instead of many), and the organizational structure is determined by user scenario, not by data flow. The diagram format expands the information available within a text-based use-case, making connections, retries, and other work flow details explicit where they are otherwise implied in the use-case description. Similar to regular flow charts, activity diagrams also make it easy to visualize branching, failure paths, and parallel activities. Process steps are rounded rectangles, decision points are diamond-shaped, flow is indicated with arrows, and round black circles indicate starting and ending states for the use-case. Figure 5 shows an activity diagram that models the use-case in figure 4:



*Figure 5: Activity Diagram modeling use-case*

## Swim-lane Diagrams

Swim-lane diagrams are a specific type of activity diagram. Their primary purpose is to make explicit which actors are performing which actions in the use-case, placing them in context within the work flow. Each actor receives a vertical column, and the actions and decision of the actor are moved to that column. Figure 6 shows the same diagram as figure 5, with added swim-lanes for Registered User and User Interface.



*Figure 6: Swim-lane diagram modeling use-case with actors*

# Class-based modeling

Class-based modeling is a component-driven approach that deals with the structure of and relationships between components, where “component” is roughly defined as almost any noun associated with a description of the system. Class-based models go hand-in-hand with use-cases; by describing the objects required to perform the use-case, they provide a mapping between user requirements and the underlying structure of the software. Although the techniques of class-based modeling borrow heavily from the concepts of object-oriented programming, it is important to note that class-based models address a wider set of concerns than software designs, and as such, contain a distinct set of semantic differences from design classes. Nevertheless, the similarities outweigh the differences, and class-based models act as a stepping-stone to future architectural designs.

The mapping process begins with a “grammatical parse” of the use-cases [Pressman05, p. 201]. Nouns are categorized into data attributes, data objects (collections of data attributes), collections of data objects, external systems, sensors, devices, people, organizations, internal roles, locations, and events are categorized. Anything in the categorization that is not a data attribute is a potential “analysis class”. Each potential class is reviewed for inclusion in the model using a decision-making process that very closely resembles object-oriented design (e.g., the item contains multiple data attributes and a reusable set of verbs apply to those attributes). After a class is identified and attributes specific to the class are added to it, related verbs are organized into ‘responsibilities’ (private operations on attributes) and ‘collaborations’ (public associations between classes) [Pressman05, p. 207].

Class diagrams and Class-Responsibility-Collaborator (CRC) cards facilitate each step of this process, as well as providing visualization of more complex class relationships. Each one approaches the problem from a slightly different perspective.

## UML Class Diagrams

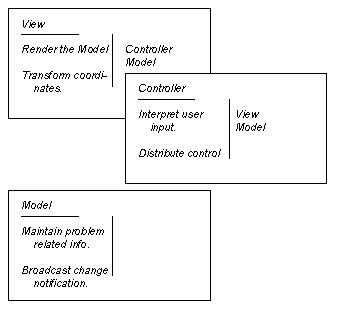
The purpose of a UML class diagram is to model the attributes and operations inside a class and the relationships between classes. Classes are represented as simple rectangles containing text, with lines to indicate relationships. There are many different types of relationships between classes. Classes can contain other classes (composition), inherit from other classes (inheritance/generalization), contain many instances of another class (aggregation), or implement a standard interface (realization) [Martin00]. UML class diagram provide mechanisms for modeling each of these relationships, as well as one-to-many, many-to-one, and other specialized relationships.

UML class diagrams are used for both requirements analysis and design analysis. In the requirements analysis phase, no implementations (private operations) are modeled. Figure 7 shows a UML class diagram of the use-case provided earlier, containing basic attributes, operations, and relationships.

*Figure 7: UML class diagram*

## Class-Responsibility-Collaborator (CRC) Cards

Class Responsibility-Collaboration cards are a series of real or virtual cards that describe classes. The purpose of this modeling technique is to identify class functionality and group it according to “responsibilities” (actions performed on attributes) within the class, and “collaborators” (other classes) outside the class that use the class in some way. Each card consists of a class name, a list of responsibilities, and a list of collaborators. [Pressman05, pp. 208-210]. Constructing CRC cards helps to identify missing operations and attributes, to define which class is responsible for which action, and to describe which other classes have an interest in this class. The existence of a collaborating object implies the a request for an attribute or operation that must be supported by the class. Figure 8 shows an example of the model-view-controller design pattern, rendered as a set of CRC cards.



*Figure 8: CRC cards*

# Behavioral Modeling

Behavioral modeling deals with the dynamic elements, the “behavior”, of a system. In a behavioral model, specific events (exchanges of information) drive a sequence of interaction between the system and one or more external entities [Pressman05, p. 217]. More than any other modeling approach, behavioral modeling blurs the line between analysis and design; it is sometimes impossible to model system state without referencing specific functionality that influences the state.

Behavioral models contain two different state representations, the internal state of the system and the observable state of the UI. Internal system state consists of both the values contained within attributes and a more abstract concept represented by the individual functionality of the class. For example, classes responsible for authentication have “not authenticated”, “authenticating”, and “authenticated” states [Pressman05, p. 218].

While multiple techniques of behavioral modeling are commonly used, including UML state diagrams, UML sequence diagrams, and control flow models, this whitepaper discusses only UML sequence diagrams.

## UML Sequence diagrams

Sequence diagrams represent the flow of events between objects over time [Pressman05, p. 220]. They are useful for observing the conversation between objects and the corresponding system state. Key classes appear as rectangular boxes, with vertical dashed lines indicating the lifespan or “lifeline” of the object. Events are shown by arrows, and vertical rectangles show time spent in processing. System states are displayed as comments along the object lifeline. Figure 9 shows a very simple sequence diagram that models an authentication scheme that results in the user’s homepage being displayed.



*Figure 9: Sequence diagram*

The semantics of sequence diagrams, i.e., calls to operations within specific classes that invoke a specific state, are more of design than of requirements analysis. However, developing a clear understanding of system state and sequence flow can help identify missing requirements where no other technique will uncover them.

# Summary

The seven techniques for analyzing requirements presented in this whitepaper are the product of thousands of man-hours and multiple decades of software engineering. Combined, they provide an excellent selection of approaches for refining and analyzing stakeholder requirements with enough flexibility to support nearly any programming language or philosophy.

**References:**

Pressman05: Roger S. Pressman, *Software Engineering: A Practitioner’s Approach, 6th Ed.* (New York: McGraw-Hill, 2005)

Propeller beanie image: <http://www.hatsinthebelfry.com/Merchant2/graphics/00000001/535full.jpg>

WIKI07a: *Data Flow Diagrams*, <http://en.wikipedia.org/wiki/Data_flow_diagram>

Martin00: Robert C. Martin, *UML Class Diagrams*, (ObjectMentor: <http://www.objectmentor.com/resources/articles/umlClassDiagrams.pdf>)

CRC cards example: <http://c2.com/doc/oopsla89/fig2.gif>