Evolving a Set of Techniques for OO Inspections

Forrest Shull[‡] **Guilherme H. Travassos**[1†] **Jeffrey Carver**[†] **Victor R. Basili**[†‡]
fshull@fc-md.umd.edu travassos@cs.umd.edu carver@cs.umd.edu basili@cs.umd.edu

[†]Experimental Software Engineering Group
Department of Computer Science
University of Maryland at College Park
A.V. Williams Building
College Park, MD 20742
USA

[‡]Fraunhofer Center - Maryland
3115 Ag/Life Sciences Surge Bldg. (#296)
University of Maryland
College Park, MD 20742
USA

Abstract

Inspecting OO designs is an important way of ensuring the quality of software under development. When high-level design activities are finished, the design documents can be inspected to verify whether they are consistent among themselves and whether the software requirements were correctly and completely captured. This paper discusses some issues regarding the definition and application of reading techniques (i.e. procedural guidelines that can be given to inspectors) to inspect high-level OO design documents. An initial set of OO Reading Techniques and their experimental evaluation is described. A method for evaluating the reading techniques in more detail, i.e. Observational Techniques, is then presented, and experiences with its use are discussed. Through these discussions, we show how the reading techniques have evolved in response to empirical evidence (both qualitative and quantitative) regarding their use in practice. The complete and current set of techniques can be found in the appendices.

I. Introduction to Reading Techniques for OO Inspections

Problem Statement

Throughout the software lifecycle many artifacts are produced other than the final application, such as requirements, design, source code, and test plans. Each of the artifacts must undergo some type of examination at different steps in the lifecycle in order to accomplish important software engineering tasks, such as verification and validation, maintenance, evolution, and reuse. Thus software reading, i.e. the process of understanding a software document in order to accomplish a particular task, is an important activity in software development. However, software reading is often done in an *ad hoc* or unstructured way; developers are taught how to write intermediate artifacts but rarely how to read and analyze them effectively. This is a problem because reading expertise is built up only slowly, through personal experience that may not be helpful or easy to communicate from one reader to the next.

Overview of our Solution

Recent research into *software reading techniques* aims to improve software reading. A Reading Technique can be defined as a series of steps for the individual analysis of a textual software product to achieve the understanding needed for a particular task. This definition has 3

[1] On leave from the Federal University of Rio de Janeiro – COPPE/Computer Science and System Engineering Department partially supported by CAPES – Brazil.
This work was partially supported by UMIACS and by NSF grant CCR9706151.

main parts. First, the *series of steps* gives the reader guidance on how to achieve the goal for the technique. By defining a concrete set of steps, we give all readers a common process to work from which we can later improve based on experience. In an ad hoc, or unstructured, reading process, the reader is not given direction on how to read, and readers use their own processes. Without a standardized process, improvement of the process is much more difficult. Secondly, a Reading Technique is for *individual analysis*, meaning that the aim of the technique is to support

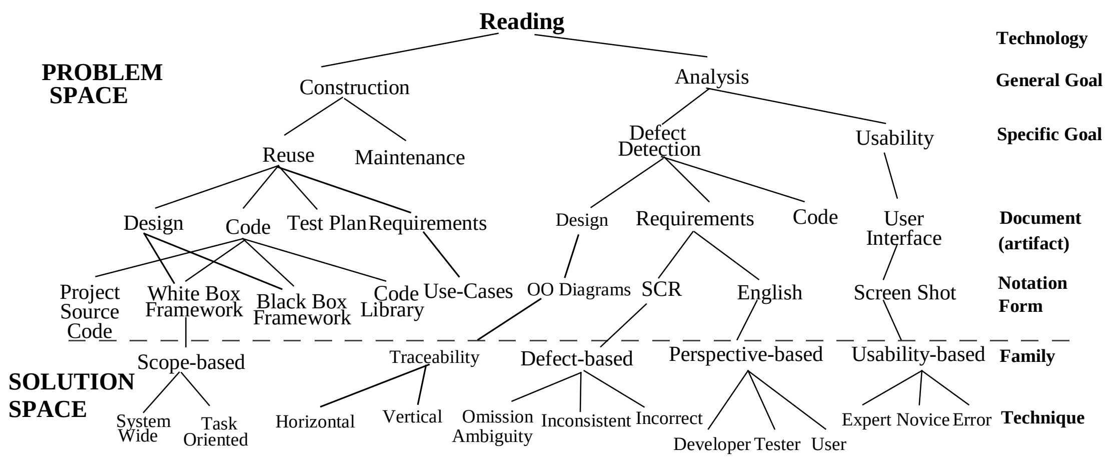


Figure 1 – Families of Reading Techniques

the understanding process within an individual reader. Finally, the techniques strive to give the reader the understanding that they need for a *particular task*, meaning that the reading techniques have a particular goal and they try to produce a certain level of understanding related to that goal [Shull98].

Families of Reading Techniques

These techniques consist of a concrete **procedure** given to a reader on what information in the document to look for. Another important component of the techniques are the **questions** that explicitly ask the reader to think about the information just uncovered in order to find defects. In previous work, we have developed families of reading techniques [Basili96]. The taxonomy of reading technique families is shown in Figure 1. The upper part of the tree (over the horizontal dashed line) models the problems that can be addressed by reading. Each level represents a further specialization of a particular software development task according to classification attributes that are shown in the rightmost column of the figure. The lower part of the tree (below the horizontal dashed line) models the specific solutions we have provided to date for the particular problems represented by each path down the tree. Each family of techniques is associated with a particular goal, artifact, and notation.

Looking back at the definition for a reading technique that was given in the previous section, we can discuss common features of the techniques. First, because each technique is a *series of steps*, it is tailorable, detailed, and specific. Secondly, because each technique is focused on a *particular task* we need the tree to organize the task space. Finally, because each technique is for *individual analysis*, certain things, such as inspection meetings, are outside the scope of the techniques. For more information on Reading Technique families see [Basili96].

Tailoring Reading Techniques

This tailorablity is an important attribute of reading techniques, by which we mean that each reading technique defined is specific to a given artifact and to a goal. For example, one specific goal could be defect detection. Software reading is an especially useful method for detecting defects since it can be performed on all documents associated with the software process, and can be applied as soon as the documents are written. Given this goal, we could imagine software reading techniques tailored to natural language requirements documents, since requirements defects (omission of information, incorrect facts, inconsistencies, ambiguities, and extraneous information) can directly affect the quality of, and effort required for, the design of a system. For this technique, the procedure would be concerned with understanding what information in the requirements is important to verify (namely, that information which is required by downstream users of the requirements document to build the system). Questions could be concerned with verifying that information to find defects that may not be uncovered by a casual or unstructured reading.

OO Reading Techniques

In this paper we will concentrate on reading techniques for Object Oriented design documents. Object Oriented reading techniques are an interesting topic of research for a variety of reasons. As OO analysis and design are increasing in popularity, it becomes more important to tailor software development technologies to work in the Object Oriented world. Because Object Oriented designs are quite different from structured designs, new Reading Techniques must be developed to address the specific issues that arise. Additionally, work in this area combined with previous research can help illuminate high-level questions in reading technique research.

The tree in Figure 1, which was described earlier, shows other work that we have done in reading. Previous research has shown that reading software artifacts to detect defects is worthwhile and possible. This defect detection has been shown in the reading of requirements both natural language requirements [Shull98] and requirements written in formal notation [Porter95] as well as reading for usability defects [Zhang99]. We have also shown that software reading is possible in the Object Oriented world, by developing techniques to read Object Oriented code and design for reuse [Basili98]. In the work we describe here, we extend that prior knowledge by building techniques that allow us to read Object Oriented designs in order to detect defects.

An OO design is a set of diagrams concerned with the representation of real world concepts as a collection of discrete objects that incorporate both data structure and behavior. Normally, high-level design activities start after the software product requirements are captured. So, concepts must be extracted from the requirements and described using the paradigm constructs. This means that requirements and design documents are built at different times, using a different viewpoint and abstraction level. When high-level design activities are finished, the documents (basically a set of well-related diagrams) can be inspected to verify whether they are consistent among themselves and if the requirements were correctly and completely captured [Travassos99a].

In order to do this we are interested in defining reading techniques that can be applied to high-level Object Oriented designs. We think that it is very important to ensure the quality of the high-level design for a few reasons. First, by focusing the techniques on the high-level design, we are ensuring that developers understand the problem fully before trying to define the solution,

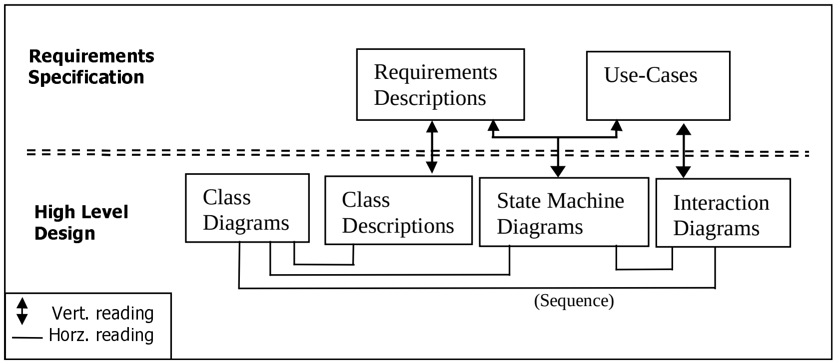


Figure 2 – Reading Techniques used in the experiment

which will appear in the low-level design. Secondly, it is important to locate and remove as many defects as possible at the higher level, because they become more difficult and more expensive to fix if they are allowed to filter down into the low-level design, or even the code [Pressman97, Pfleeger98].

II. Evolution of the Techniques

Since we could find little work done in research or practice on effective reviews of OO work products, our investigation into this area was necessarily more exploratory than confirmatory. That is, we have not been seeking to verify or disprove well-formulated hypotheses, since this area is not sufficiently well understood for us to be able to have confidence in the usefulness or practicality of specific hypotheses. Instead our studies have focused on developing an understanding, based on observation and analysis rather than theory, of the real issues and problems involved in OO reviews. We aim to use the information gained from such empirical study to build up tentative but reasonable hypotheses, which can later be tested to good effect in more rigorous studies. This early cycle of observation, feedback, and improvement of our understanding is necessary so that confirmatory work in this area will be well-grounded in practical considerations.

With this information as a background we have developed two main questions which drive all of the Object Oriented reading techniques.

1) Do all of the design artifacts describe the same system, i.e. are the individual artifacts consistent with each other?
2) Do the design artifacts, as a whole, accurately describe the same system that the requirements describe, i.e. are the requirements and the design consistent?

1. Defining the Initial Techniques

With the ideas discussed earlier and the two main questions in mind, we developed an initial set of reading techniques. We chose to focus the reading techniques on assuring the consistency of information that could be traced between multiple documents. Using the two questions from the last section, we were able to define two different categories of techniques. The first question strives to explore whether all of the artifacts from one lifecycle phase (the

High-level design) are consistent with each other. Because these techniques stay within the same phase for comparing artifacts, we have called these techniques horizontal techniques. The second question helps us to determine if the artifacts from one lifecycle phase, the requirements (represented by their description and possible scenarios), were accurately captured by the artifacts from another lifecycle phase, the High-level design. Because the documents that are compared here are from different lifecycle phases, we have called these techniques vertical techniques.

Using our previous knowledge that a reading technique should be a series of steps to help the reader focus on the right information, we came up with a set of procedural techniques for the comparison of the artifacts. Figure 2 illustrates all of the techniques that we have defined, also showing which are horizontal and which are vertical. For each group of documents that was to be compared, using knowledge of Object Oriented design concepts, we came up with a set of steps that would focus the reader on information that should be found in both documents. Associated with each step was a set of questions that the reader could use to help them locate possible defects. One of the initial techniques can be seen in Figure 3.

Now that we had an initial set of Reading Techniques, we needed to evaluate them. In order to do that, we designed and carried out an experiment.

Reading Technique for Class diagrams x Class descriptions

For each class modeled into the class diagram, do:

1) Read the **class description** to find an associated description to the class.

☞ Underline with a yellow pen the portion of the Class descriptions corresponding to the class

Verify if:

1.1) All the attributes are described and with basic types associated

☞ Underline them with a blue pen

1.2) All the behaviors and conditions are described

☞ Underline them with a green pen

1.3) All the class inheritance relationships are described

☞ Draw a box around them with a yellow pen if there is any

1.4) All the class relationships (association, aggregation and composition) are described with multiplicity indication

☞ Circle each multiplicity indication with a blue pen if it is correct.

Note that the emphasis is on syntactic checking, that is, that the OO notation on both diagrams agrees.

Figure 3 - First version of a horizontal reading technique.

2. Experimental evaluation of Reading techniques

Method

The main goal of this experiment was to evaluate the feasibility of applying reading techniques to an Object Oriented design. A secondary goal was to receive feedback on the format and content of the reading techniques, which was used to create a second version of the techniques.

Subjects

The subjects in this experiment came from an undergraduate software engineering course at UMCP during the Fall 1998 semester. The 44 students in the class had a mix of previous software experience: 32% had some previous industry experience in software design from requirements and/or use cases, 9% had no prior experience at all in software design, and the remaining majority of the class (59%) had classroom experience with design but had not used it on a real system. However, all students were trained in OO development, UML and OO software design activities as a part of the course. The subjects were randomly assigned into 15 teams (14 teams with 3 students each and 1 team with 2 students) for the experiment.

Materials

The materials under study during this experiment consisted of the initial version of the reading techniques. They were applied to the design of a "Loan Arranger" system responsible for organizing the loans held by a financial consolidating organization, and for bundling them for resale to investors. It was a small system, but contained some design complexity due to non-functional performance requirements. Table 1 summarizes the size of this design by reporting for each class the number of attributes, Weighted Methods/Class (WMC), Depth of Inheritance (DIT), Number of Children (NOC), and Coupling Between Objects (CBO). Additionally, there were 3 state diagrams and 5 sequence diagrams (Seq1-Seq5); the classes participating in each are marked.

Class Name	Attrs.	WMC	DIT	NOC	CBO	State Dgm. exists?	Seq1 Contains?	Seq2 Contains?	Seq3 Contains?	Seq4 Contains?	Seq5 Contains?
Property	5	0	0	0	1						
Borrower	2	2	0	0	1						
Lender	3	1	0	0	2		yes				
Loan	3	3	0	2	4	yes				yes	
Fixed Rate Loan	0	1	1	0	4						
Adjustable Rate Loan	0	1	1	0	4						
Bundle	5	2	0	0	2	yes				yes	
Investment Request	4	1	0	0	1					yes	
Loan Arranger	0	15	0	0	4		Yes	yes	yes	yes	yes
Financial Org.	1	0	0	0	2	yes	Yes	yes	yes	yes	yes
Loan Analyst	1	12	0	0	3		yes	yes	yes	yes	yes

Table 1 – Size measures for the inspected design.

Procedure

The first thing that the subjects did was to receive the Loan Arranger requirements and perform an inspection. This served the purpose of improving the understanding of the system. After this, the subjects were given the requirements and a set of use cases and asked to create an OO design of the system. Once the designs were complete, we selected two of the best designs submitted. Each team was given one of the designs to inspect with the reading techniques. (Two designs were chosen to ensure that no team inspected their own design. But, to keep the results comparable, all but one team reviewed the same design.)

We should note here that we had no control group; that is, we could not compare the subject's effectiveness to their own effectiveness using another OO inspection method. There were two main reasons for this decision. The first is that we are aware of no other published methods for reading OO designs with which we could compare. The second reason is that because we were in a classroom environment, it was not possible to teach only a portion of the class the reading techniques and use the others as a control.

Each team applied the whole set of techniques, but the techniques were divided up in such a way that each subject had to deal only with a small number. One subject performed the vertical reading, while the other two divided the horizontal techniques between them. After performing their individual reviews, the team members met to compile their individual defect lists into a final list that reflected the group consensus.

As was stated earlier, the main goal of this experiment was to evaluate the feasibility of OO reading techniques. In order to do this, we collected a number of metrics, both qualitative and quantitative. Table 2 shows a subset of the metrics that were collected. The qualitative data were collected using questionnaires, which were collected with the defect lists, retrospective questionnaires at the end of the semester, and post-hoc interviews conducted separately for each team. By collecting the data at various times throughout the process we were able to check the consistency of the subjects' answers, and have more confidence in their accuracy.

We collected quantitative data by having the subjects turn in their defect lists. Because this was mainly a feasibility study, we did not discriminate between true defects reported and false positives. That is, we made the assumption in our counting of defects that all defects reported were real problems with the document.

When Collected	Metrics
Before the study	a) Details on subjects' amount of experience with requirements, design, and code
After individual review	b) Time spent on review (in minutes) c) Opinion of effectiveness of technique (measured by what percentage of the defects in the document they thought they had found) d) How closely they followed the techniques (measured on a 3-point scale) e) Number and type of defects reported
After team meeting	f) Time spent (in minutes) g) Usefulness of different perspectives (open-ended question)
In post-hoc interviews	h) How closely they followed technique (open-ended question, to corroborate d) i) Were the techniques practical, would they use some or all of them again (open-ended question)

Table 2 – Subset of metrics collected in the feasibility study

Results and Lessons Learned

The quantitative data from this experiment showed some positive results. Using the techniques did allow teams to detect defects (11 were reported, on average; 11.7 for horizontal readers, 10.4 for vertical), and in general subjects tended to agree that the techniques were helpful. Also, the vertical techniques tended to find more defects of omitted and incorrect functionality, while the horizontal techniques tended to find more defects of ambiguities and inconsistencies between design documents, lending some credence to the idea that the distinction between horizontal and vertical techniques is real and useful.

The qualitative data also provided us with some lessons that helped to improve the next version of the techniques. From the qualitative data we were able to learn three global lessons on how to improve the techniques for the second version. The first lesson that we learned was that **OO reading techniques should concentrate on semantic, not syntactic, issues**. Based on the qualitative data that we received from the subjects it was clear that the reading techniques should explore more of the semantic information contained in the models as opposed to the syntactic (notation) information (Figure 3 illustrates the syntactic version). When we talk about semantic checking we are referring to validation of whether design decisions make sense and are feasible. For example, a reader may be asked to check that appropriate types have been assigned to class attributes, given his or her understanding (from the requirements) of what information those attributes are supposed to represent. As another example, a reader may be given a procedure for how to check that some class (or combination of classes) is responsible for each functionality of the system, as described in the requirements. This new semantic version of the techniques required the reader to recreate the *rationales* that describe why a design appears the way it does, in response to the constraints and requirements set down in the requirements specification. This new version allowed the reading techniques to be less mechanical and require more thought on the part of the reader. As an example Figure 4 highlights the new version of the technique previously examined in Figure 3. Note that in the technique in Figure 4, the reader is instructed to determine if all of the attributes have been described and if their types make sense.

Reading 4 -- Class diagrams x Class descriptions

Goal: To verify that the detailed descriptions of classes contain all the information necessary according to the class diagram, and that the description of classes make semantic sense.

Inputs to Process:

1. A class diagram (possibly divided into packages) that describes the classes of a system and how they are associated.
2. A set of class descriptions that lists the classes of a system along with their attributes and behaviors.

1) **Read the class diagram to understand the necessary properties of the classes in the system.**

INPUTS: Class diagram;
Class description.

OUTPUTS: Discrepancy reports.

For **each class on the class diagram,** perform the following steps:

…

☞ Verify that all the attributes are described along with basic types.
Are the same set of attributes present in both the class description and the class diagram? If not, it means that attributes are not appropriately described. This represents an inconsistency between the documents. Fill in a discrepancy report describing this situation and showing which document didn't capture the appropriate information.
Can this class *meaningfully* encapsulate all these attributes? That is, does it make sense to have these attributes in the class description? Are the basic types assigned to the attributes *feasible* according to the description of the attribute? If not, it represents an ambiguity or an incorrect fact. Fill in a discrepancy report describing this situation and showing which document didn't capture the appropriate information.

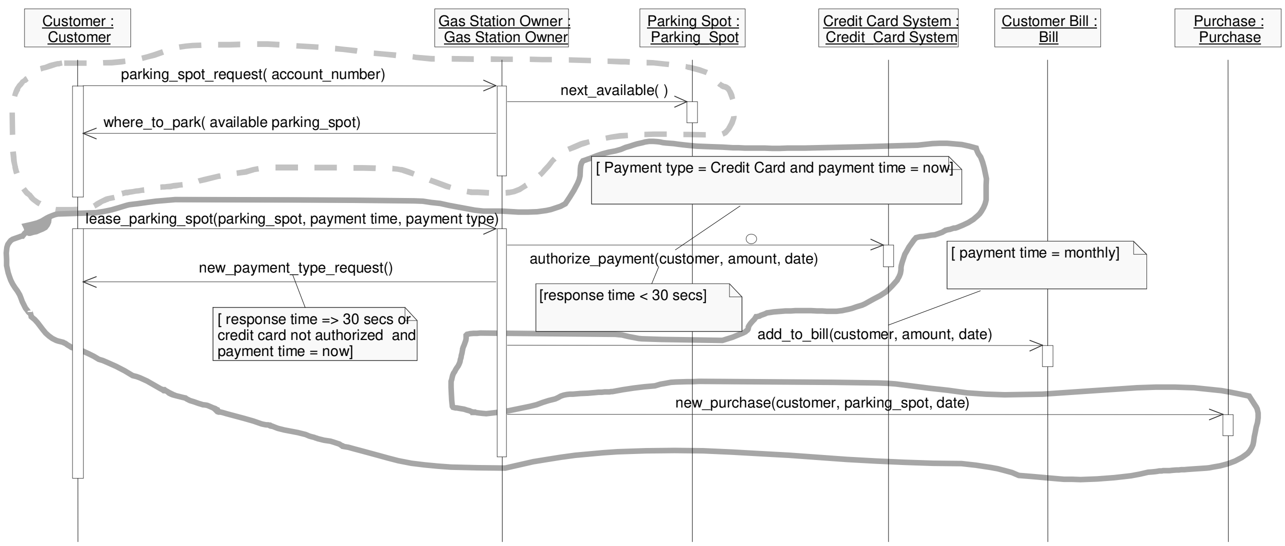
…

Note the introduction of semantic checking, that is, that the information expressed by the notation is *feasible, possible,* or *meaningful.*

Figure 4 – Second version of a horizontal reading technique.

The second lesson learned was that reading techniques need to include not only instructions for the reader, but some motivation as to why those instructions are necessary. The information in the qualitative data suggested that subjects were interested in why the steps of the technique were important and useful, as opposed to just being told what to do. It seemed that when the subjects understand the overall goal they are more likely to find value in the individual steps, which previously may have seemed of little value.

The third lesson that was learned was that the level of granularity of the instructions needs to be precisely described. Discussing functionality is a difficult but necessary part of the reading techniques. The difficulty comes from the many different levels of granularity at which system behavior can be described. To solve this problem we decided to define 3 distinct ways of discussing functionality, ranging from very specific system *messages* (the communications



Combinations of messages that form system *services* have been marked. Conditions and constraints are included as annotations on the diagram. "Response time < 30 secs" represents a nonfunctional *constraint* on the way certain functionality has to be implemented. "Payment time = monthly" is an example of a *condition* that must be true for a particular message to be executed; in this case, the system variable "payment time" must have the value "monthly."

Figure 5 - A sequence diagram capturing how classes collaborate to perform system functionality.

between objects that work together to implement system behavior) to system *services* (the steps by which some larger goal or functionality is achieved) and finally, at the top level, system *functionalities* (the behavior of the system, from the user's point of view). Thus, we can talk about specific behaviors of the classes combining to provide services of the system and, in turn, the high-level functionality that the user sees. For example consider the diagram presented in Figure 5. The sequence diagram describes how classes collaborate to provide some *functionality*: the ability to lease a parking spot. This functionality is meant to describe a use of the system from the user's point of view; although the user may have to perform several steps in his interaction with the system, we expect that his or her final goal is the lease of a spot to park his car. Two *services* are marked on the diagram, represented by the heavy dashed and solid lines, which group together a collection of *messages*. These services represent particular steps that must be accomplished for the user to achieve the task of purchasing the parking spot. The dashed grouping may be thought of as the service of "getting an open spot" while the grouping circled by the solid line accomplishes the step of "paying for the spot at the time of leasing." To the user, neither step makes sense as a goal in and of itself; e.g. it is of little use to the customer to find an open spot but not pay for it.

With the information gained from these lessons we were able to generate the next version of the reading techniques, making global changes to the techniques. What was needed now was to evaluate each technique in more detail. We needed to collect information about the individual steps in each technique as opposed to the global information that we had collected up to this point. This seemed best accomplished through the use of "observational techniques", by which we mean an experimental subject performs some task while the experimenter gathers data about what exactly the subject does. The purpose of the observation is to collect data about how the particular task is accomplished. To test the feasibility of the proposed observational studies methods and the changes to the technique format, a pilot study was run.

3. Pilot Study of Observational Techniques

Background

Observational methods are distinct from *retrospective* methods of collecting qualitative data about processes, such as post-mortem interviews or questionnaires. Observational studies address an important drawback of retrospective techniques, namely that retrospection does not present the experimenter with a completely accurate picture of the subject's thought process. This is because often, when using retrospective techniques, it is difficult for a subject to reconstruct from memory the actual thought process they went through to solve a problem. Another drawback to retrospective techniques is that if a subject is given time to reflect on what he did, then he may, intentionally or casually, present his thought process and ideas in a more structured and coherent way than they actually occurred. As our goal here is to design reading techniques that support the reader, knowledge of how the thought process actually works is important. Because of all this, it has been suggested that employing techniques to observe subjects while working may be a way to capture more complete and accurate information about what is really going on [Van Someren94].

In her paper on observational studies, Dr. Singer discusses how data collection can be split into 2 subtypes, observational and inquisitive. Observational data is collected while the process is being executed, but without direction from the researcher. For instance, subjects are told to think out loud as they execute a technique. This allows the researcher to gain insight into how the process is executed, for example, the researcher can record if the subject becomes confused or does not know what to do next at any step of the process. Collecting observational data is largely passive on the part of the researcher, since it is required that the researcher avoids interfering with the process being studied as much as possible.

Inquisitive data is collected at the completion of a process step, rather than during its execution. Data collection requires the researcher to be more assertive, since they must solicit responses to definite questions rather than observe process execution as it occurs. For example, at the end of each step, the researcher could ask the subject for qualitative feedback on the reading technique. This is not information that the subject would normally think about while executing the process, yet it is invaluable to collect at this time, while it is still fresh in the mind of the subject [Singer96].

Method

This pilot study was undertaken to evaluate the feasiblity of using observational techniques to evaluate the Object Oriented reading techniques. In this study we hoped to gain some insight into how the observational techniques interacted with the reading techniques. Our main goal was to come out of the pilot study with observational techniques that we had some confidence in, instead of testing something blindly on a large group.

Subjects

There were two subjects in this pilot study. The subjects were selected because of their experience and knowledge in the area of process improvement, so that we could receive valuable feedback on the observational techniques and the reading techniques before applying them on a larger scale.

Materials

Based on the two types of questions, inquisitive and observational, we designed a guide that the researcher would follow when observing an experimental subject. The guide consisted of a copy of the reading technique (so that researchers could follow the subject's progress) in which each step was augmented with a series of questions to be filled in as the information became available. These questions asked for both observational and inquisitive data, and were decided upon after discussion among the research team of the most important aspects of the techniques to get feedback on. The questions were concerned with how long the step took, if there was anything that was confusing about the step, and whether or not the subject knew of a better way to accomplish that step.

Procedure

Because this was only a pilot study, only two of the reading techniques were selected for evaluation. We decided that because we were most interested in improving the observational techniques and not the reading techniques, it was not important or useful in the pilot study to evaluate all of the reading techniques. Each of the subjects has been given an OO reading technique and a set of artifacts. There were two members of the research team that acted as observers. One was an active observer, following the guide discussed in the Materials section. This observer was there to: 1) step the subject through the procedure; 2) remind the subject to "think out loud", if necessary, so that the observational data could be collected; and 3) to capture information about the circumstances in which the subject experiences problems or has trouble understanding the OO reading technique. He also took note of the time consumed by each step of the process and whether or not the step was successful in finding defects. The other observer was a passive one. This observer wrote down observations about what was going on, but did not interfere with the reading process. This allowed a more objective set of notes about the study method to be collected.

Results and Lessons Learned

First we will consider the results gained from the observational data. These results deal mainly with the amout of time taken for each step. We observed that different aspects influenced the time it took to apply such techniques. The prior experience of the reader and the complexity of the information being inspected seem to be influential factors, although sometimes the order of steps and the abstraction level of the reading technique contributed to increase the time required for applying the techniques. For example, the first time that a particular reader used the techniques, extra time was required to understand the steps and even to understand the types of information in the inspected document that were relevant. But, after the reader applied the techniques once, the time spent to apply such techniques diminished. The observational techniques revealed several causes for this, including that the reader could better understand the steps and, interestingly, could mentally reorganize some of the steps in an order that made more sense for him.

It was also observed that the complexity of the class being inspected is an important factor that can determine how long particular steps will take. This aspect was noticed especially when readers tried to read classes belonging to inheritance hierarchies. In this situation, it was also observed that readers have a tendency to apply a top-down rather than a bottom-up approach. In other words, to understand the real-world concept or rationale behind the construction of the design, readers typically need to build the whole conceptual model before understanding the basic concept.

Next we consider the results gained from the inquisitive data. These results deal mainly with the influence of prior knowledge. Some domain knowledge seemed to be necessary to support horizontal reading. Readers suggested that they didn't need to have the whole set of requirements to read design artifacts, but that some description of the system, its main concepts and objectives, would be worthwhile. This information could be useful to justify the design choices or rationales somehow, with the goal of easing the identification of the concepts captured and used to represent the high-level system solution. Another point reported by the readers regarded the necessary level of object-oriented development expertise. Although the techniques were meant to be described to allow any reader to read design artifacts[2], readers seemed to be comfortable when they knew the basic concepts of the OO paradigm. For instance, the techniques provided examples and definitions described with enough detail to allow the reader to understand the idea behind each one of the constructs and concepts used. But, when different levels of abstraction were used (e.g., references to objects versus classes), readers without a more detailed knowledge of the object-oriented paradigm encountered difficulty. It should be noted that this does not imply that readers need experience in OO development, just some knowledge of the basic OO concepts such as class, object, attribute, and behavior.

Another interesting observation was that readers typically knew of no other approach or tool that could elicit the types of information being checked by the reading techniques. The sole exception was for some of the steps of the horizontal reading techniques (mainly the ones dealing

[2] We assumed that readers could use the techniques without previous detailed knowledge about the OO constructs. This assumption is important mainly when vertical reading is been applied, because it allows the participation of readers who have knowledge of the system being constructed but not in-depth knowledge of software construction (e.g. system users or customers).

with syntactical issues), for which readers suggested using a CASE tool to automate the consistency check.

Lastly we will examine the problems that we encountered in the pilot study that led us to improvements in the techniques and associated artifacts. We were able to identify three basic types of problems that were present in this version of the techniques: organizational issues, semantic issues, and the format of software artifacts being inspected.

Organizational problems are concerned with the way that the readers should apply the different reading techniques. Sometimes, readers suggested that applying one or another step before (or after) others could facilitate the identification of some features or information in the inspected document. For example, it was suggested that it is easier to understand the attributes of a class after the set of behaviors has been understood. This type of suggestion was possible only after readers had applied the techniques for the second or third time, having better understood them and having mentally reorganized the steps necessary to accomplish the tasks.

Another type of organizational problem regarded the format used to report defects. Readers uncovered several problems with using the defect taxonomy (Table 3) to report defects. Some confusion concerned the level of granularity that should be used. For instance, some students experienced difficulties trying to describe how a class description could be ambiguous when actually the ambiguity was related to only one attribute of the class.

Some semantic problems were also identified with the new reading technique version. Basically, these were concerned with how a reader’s domain knowledge affects the execution of the techniques, and how questions should be phrased as part of the techniques to force readers to think about the semantic content of the information they were reading. Readers who used horizontal reading techniques reported that having some domain knowledge could speed up the reading process, allowing them to quickly recognize some concepts captured by the design diagrams. This situation was not detected in the vertical reading techniques, probably because readers were checking the design against the requirements description, which already aims to describe the necessary knowledge to understand the problem.

Type of Defect	***Description***
Omission	One or more design diagrams that should contain some concept from the general requirements or from the requirements document do not contain a representation for that concept.
Incorrect Fact	A design diagram contains a misrepresentation of a concept described in the general requirements or requirements document.
Inconsistency	A representation of a concept in one design diagram disagrees with a representation of the same concept in either the same or another design diagram.
Ambiguity	A representation of a concept in the design is unclear, and could cause a user of the document (developer, low-level designer, etc.) to misinterpret or misunderstand the meaning of the concept.
Extraneous Information	The design includes information that, while perhaps true, does not apply to this domain and should not be included in the design.

Table 3 – Types of software defects, and their specific definitions for OO designs

Last, but not least, subjects asked about the format of the artifacts being inspected. Although UML notation had been used during the construction process, specific changes to the format could be proposed to facilitate the reading process. As suggested by the readers, some organizational improvements (e.g. having an index to find the concepts quickly) could speed up searching consequently providing a way to speed up the reading process [Travassos99b].

III. Conclusions

The object oriented reading techniques (OORTs) have been, and still are, evolving since their first definition. New issues and improvements have been included based on the feedback of readers and volunteers. Throughout this process, we have been trying to capture new features and to understand whether the latest reading technique version keeps its feasibility and interest. We have found observational techniques useful, because they allowed us to follow the reading process as it occurred, rather than trying to interpret the readers' post-hoc answers as was done before. Observing how readers normally try to read diagrams challenged many of our assumptions about how our techniques were actually being applied.

For now, a new and reorganized version of the OORTs, dealing with more semantic issues, has been created in an attempt to facilitate the reading process (see Appendix B).

References

[Basili96] V. Basili, G. Caldiera, F. Lanubile, and F. Shull. Studies on Reading Techniques. In *Proc. of the Twenty-First Annual Software Engineering Workshop*, SEL-96-002, pages 59-65, Greenbelt, MD, December 1996.

[Basili98] V. Basili, F. Lanubile, and F. Shull. Investigating Maintenance Processes in a Framework-Based Environment. *Proc. Of the International Conference on Software Maintenance*, Bethesda, MD. November 1998.

[Pfleeger98] Shari Lawrence Pfleeger, *Software Engineering: Theory and Practice*, Prentice-Hall, 1998.

[Porter95] A. Porter, L. Votta Jr., V. Basili. Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment. *IEEE Transactions on Software Engineering*, 21(6): 563-575, June 1995.

[Pressman97] Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, Fourth edition, McGraw-Hill, 1997.

[Shull98] F. Shull. *Developing Techniques for Using Software Documents: A Series of Empirical Studies*. Ph.D. thesis, University of Maryland, College Park, December 1998.

[Singer96] J. Singer and T. Lethbridge. "Methods for Studying Maintenance Activities." In *Proc. of the Workshop for Empirical Studies of Software Maintenance*, 1996

[Travassos99a] G. H. Travassos, F. Shull, M. Fredericks, V. R. Basili**.** Detecting Defects in Object Oriented Designs: Using Reading Techniques to Increase Software Quality**,** OOPSLA'99, Denver, Colorado, USA, November 1999

[Travassos99b] G. H. Travassos, F. Shull, and J. Carver. "Evolving a Process for Inspecting OO Designs." XIII SBES: *Workshop on Software Quality*. Florianópolis, Curitiba, Brazil, October 1999.

[Van Someren94] M.W. Van Someren, Y.F. Barnard, and J.A.C Sandberg. *The Think Aloud Method: A Practical guide to Modeling Cognitive Processes*. Academic Press: London. 1994.

[Zhang99] Z. Zhang. *The Design and Empirical Studies of Perspective-Based Usability Inspection*. PhD. Thesis, University of Maryland, College Park, June 1999.

APPENDIX A - The Definitions

Throughout the description of the techniques, the following terms are constantly used:

1. Functionality: *Functionality* describes the behavior of the system. Typically, functionality is described from the user's point of view. That is, a description of system functionality should answer the question: What can a user use the system to do? In the case of a word processor, an example of system functionality is formatting text.

2. Service: Like "functionality", a *service* of the system is an action performed by the system. However, services are much more low-level; they are the "atomic units" out of which system functionalities are composed. Users do not typically consider services an end in themselves; rather, services are the steps by which some larger goal or functionality is achieved. In the case of a word processor, typical services include selecting text, using pull-down menus, and changing the font of a selection.

3. Message: *Messages* are the very lowest-level behaviors out of which system services and, in turn, functionalities are composed. They represent the communication between objects that work together to implement system behavior. Messages may be shown on sequence diagrams and must have associated class behaviors.

For example, consider the example diagrams provided in the appendix D. In example 2, the sequence diagram describes how classes collaborate to provide some *functionality*: the ability to lease a parking spot. This functionality is meant to describe a use of the system from the user's point of view; although the user may have to perform several steps in his interaction with the system, we expect that his or her final goal is the lease of a spot to park his car.

Two *services* are marked on the diagram, represented by the heavy dashed and solid lines, which group together a collection of *messages*. These services represent particular steps that must be accomplished for the user to achieve the task of purchasing the parking spot. The dashed grouping may be thought of as the service of "getting an open spot" while the grouping circled by the solid line accomplishes the step of "paying for the spot at the time of leasing." To the user, neither step makes sense as a goal in and of itself; e.g. it is of little use to the customer to find an open spot but not pay for it.

It should be noted that there may be multiple ways to group messages together into services. The messages lease_parking_spot, add_to_bill, and new_purchase may be grouped to compose a service that can be thought of as "paying for a spot via monthly bill." Each of these services represents a different execution path the system will follow under different conditions, and thus all are necessary to describe the full range of system functionality. In some cases, the designer may choose to use a number of similar sequence diagrams, with each diagram showing one such execution path, in order to avoid the complexity of many services being represented on the same diagram, as is the case in Example 2.

APPENDIX B - The Techniques
B.1 Reading 1 -- Sequence x Class Diagrams

Goal: To verify that the class diagram for the system describes classes and their relationships in such a way that the behaviors specified in the sequence diagrams are correctly captured. To do this, you will first check that the classes and objects specified in the sequence diagram appear in the class diagram. Then you will check that the class diagram describes relationships, behaviors, and conditions that capture the dynamic services as described on the sequence diagram.

Inputs to process:

1. A class diagram (possibly divided into packages) that describes the classes of a system and how they are associated.
2. Sequence diagrams that describe the classes, objects, and possibly actors of a system and how they collaborate to capture services of the system.

1) **Take a sequence diagram and read it to understand the system services described and how the system should implement those services.**

INPUTS: Sequence diagram (SD).

OUTPUTS: System objects (marked in blue on SD);
Services of the system (marked in green on SD);
Conditions on the services (marked in yellow on SD).

- ☞ For each sequence diagram, underline the system objects and classes, and any actors, with a blue pen.
- ☞ Underline the information exchanged between objects (the horizontal arrows) with a green pen. Consider whether this information represents *messages* or *services* of the system. If the information exchanged is very detailed, at the level of messages, you should abstract several messages together to understand the services they work to provide. Example 2 provides an illustration of messages being abstracted into services. Annotate the sequence diagram by writing down these services, and underline them in green also.
- ☞ Circle any of the following constraints on the messages and services with a yellow pen: restrictions on the number of classes/objects to which a message can be sent, restrictions on the global values of an attribute, dependencies between data, or time constraints that can affect the state of the object. Also circle any conditions that determine under what circumstances a message will be sent. The sequence diagram in Example 2 contains several examples of constraints and conditions on messages. The conditions concerning payment type and payment time determine when messages authorize_payment and new_payment_type_request will be sent , while the restrictions on response_time for message authorize_payment represent time constraints.

2) **Identify and inspect the related class diagrams, to identify if the corresponding system objects are described accurately.**

INPUTS: Sequence diagrams, with objects, services, and constraints marked;
Class diagrams.

OUTPUTS: Discrepancy reports.

- ☞ Verify that every object, class, and actor used in the sequence diagram is represented by a concrete class in a class diagram. For classes and actors, simply find the name on the class diagram. For objects, find the name of the class from which the object is instantiated. **If a class or object cannot be found on the class diagram, it means that the information is inconsistent between both documents. If an actor cannot be found, it may also mean that there is inconsistent information; you need to consider whether the actor must be represented as a class in the system in order to provide necessary behavior. Fill in a discrepancy report describing these problems.**
- ☞ Verify that for every green-marked service or message on the sequence diagram, there is a corresponding behavior on the class diagram. Verify that there are class behaviors in the class diagram that encapsulate the

higher-level services provided by the sequence diagram. To do this, make sure that the class or object that *receives* the message on the sequence diagram, or should be responsible for the service, possesses an associated behavior on the class diagram. Also make sure that there exists some kind of association (on the class diagram) between the two classes that the message connects (on the sequence diagram). Remember that in both cases, you may need to trace upwards through any inheritance trees in which the class belongs to find the necessary features. Finally, verify that for each service, the messages described by the sequence diagram are sufficient to achieve that service.

Is there a message on the sequence diagram for which the receiving class does not contain an appropriate behavior on the class diagram? If yes, it means that there is an inconsistency between the diagrams. The sequence diagram implies that a behavior must exist, but no such behavior is recorded for the appropriate class on the class diagram. Fill in a discrepancy report describing the problem. Are there appropriate behaviors for the system services? If not, it means that no class assumes responsibility for a particular service. Fill in a discrepancy report describing the problem. Is there an association on the class diagram between the two classes between which the message is sent? If not, necessary information has been omitted from the class diagram. If a message is exchanged between two classes they must be associated in some way. Fill in a discrepancy report describing the problem.

Are there any missing behaviors, without which the system service cannot be achieved? If so, it means that there is an omission from the sequence diagram. Fill in a discrepancy report describing the problem.

- ☞ Verify that the constraints identified in the sequence diagram can be fulfilled according to the class diagram. Consider the following cases: 1) If the sequence diagram places restrictions on the number of objects that can receive a message, make sure that constraint appears as cardinality information for the appropriate association in the class diagram. 2) If the sequence diagram specifies a range of permissible values for data, make sure that constraint appears as a value range on an attribute in the class diagram. 3) If the sequence diagram contains information concerning the dependencies between data or objects (e.g. “a ‘Bill’ object cannot exist unless at least one ‘Purchase’ object exists”) make sure that this information is included as a constraint on a class or relation on the class diagram. Dependencies between objects may also be represented by cardinality constraints on relationships. 4) If the sequence diagram contains timing constraints that could affect the state of an object (e.g. “if no input is received within 5 minutes then the window should be closed”) make sure that this information is included as a constraint on a class or relation on the class diagram. For example, the class diagram in Example 3 contains a timing constraint for the class “Credit_Card_System” since it applies to all instantiations of this class. The conditional expressions from Example 2 should not appear in the class diagram because they do not affect the state of a class.

  Could you find the data but they do not conform to the behavior arguments? Or, could you find the constraints but they do not completely agree in both documents? If yes, it means that the documents are inconsistent. Fill in a discrepancy report to describe the problem.

- ☞ Finally, for each class, message, and data identified above, think about whether, *based on your previous experience,* it results in a reasonable design. For example, think about quality attributes of the design such as cohesion (do all the behaviors and attributes of a class really belong together?) and coupling (are the relations between classes appropriate?).

  Does it make sense for the class to receive this message with these data? Could you verify if the constraints are feasible? Are all of the necessary attributes defined? If not, the diagrams may contain incorrect facts. Fill in a discrepancy report to describe the problem. For the classes specified in the sequence diagram, do the behaviors and attributes specified for them on the class diagram make sense? Is the *name* of the class appropriate for the domain, and for its attributes and behaviors? Are the relationships with other classes appropriate? Are the relationships of the right *type*? (For example, has a composition relationship been used where an association makes sense?) If not, you have found an incorrect fact because something in the design contradicts your knowledge of the domain. Fill in a discrepancy report to describe the problem.

B.2 Reading 2 -- State diagrams x Class description

Goal: To verify that the classes are defined in a way that can capture the functionality specified by the state diagrams.

Inputs to Process:

1. A set of class descriptions that lists the classes of a system along with their attributes and behaviors.
2. A state diagram that describes the internal states in which an object may exist, and the possible transitions between states.

For **each state diagram,** perform the following steps:

1) **Read the state diagram to understand the possible states of the object and the actions that trigger transitions between them.**

   INPUTS: State diagram (SD).

   OUTPUTS: Object States (marked in blue on SD);
   Transition Actions (marked in green on SD);
   Discrepancy reports.

   ☞ Determine which class is being modeled by this state diagram.

   Could you identify the type of the object that this state machine is modeling? If you can't find this information, you have found a defect of omission. Fill in a discrepancy report for this.

   ☞ Trace the sequence of states and the *transition actions* (system changes during the lifetime of the object, which trigger a transition from one state to another) through the state diagram. Begin at the start state (filled circle) and follow the transitions until you reach an end state (double circle). Make sure you have covered all transitions.

   ☞ Underline the name of each state, as you come to it, with a blue pen.

   ☞ Highlight transition actions (represented by arrows) as you come to them using a green pen. For example, the state diagram provided in Example 5 contains seven transition actions. The arrow leading from the state labeled "authorizing" back to itself represents an action that does not actually change the state of the object.

   ☞ Think about the states and actions you have just identified, and how they fit together.

   Is it possible to understand and describe what is going on with the object just by reading this state machine? If not, you may have discovered a defect of ambiguity; the behavior of this class over its lifetime is not well described.

2) **Find the class or class hierarchy, attributes, and behaviors on the class description that correspond to the concepts on the state diagram.**

   INPUTS: Class description (CD);
   Object States (marked in blue on SD);
   Transition Actions (marked in green on SD).

   OUTPUTS: Relevant object attributes (marked in blue on CD);
   Relevant object behaviors (marked in green on CD);
   Discrepancy reports.

   ☞ Use the class description to find the class or class hierarchy that corresponds to this state diagram. **Did you find the corresponding class? If not, you have found a defect of inconsistency. The state machine describes a class that has not been described on the class description.**

   ☞ Find how the responsible class encapsulates the blue-underlined states described on the state diagram. States may be encapsulated:

- 1 attribute explicitly. (An attribute exists whose possible values correspond to system states, e.g. attribute “mode” with possible values “on”, “off”.)
- 1 attribute implicitly. (An object is considered to be in a specific state depending on the value of some attribute, but the state is not recorded explicitly. E.g. if a>5 the object behaves one way, for other values of a another behavior is appropriate, but nothing explicitly records the current state.)
- a combination of attributes.
- class type. (E.g. subclasses “fixed rate loan” and “variable rate loan” can be considered states of parent class “loan”.) Remember to check the corresponding class and all parents in its inheritance hierarchy. Mark each blue-underlined state with a star (*) when it is found.

  Has the class captured the idea of the modeled states? If not, you may have found a defect of inconsistency; the state diagram specifies certain states for an object that cannot be captured by the class as it is described. Or, you may have found an ambiguity; it is not clear how the modeled states can be captured by a class.

☞ For each green-highlighted transition action on the state diagram, verify that there are class behaviors capable of achieving that transition. Remember to look both in the currently selected class and any classes higher in the inheritance hierarchy.

(Keep in mind the following possible exceptions: 1) The transition depends on a global attribute, outside of the class hierarchy. 2) In instances of poor design, i.e. high coupling and public class attributes, behaviors in associated classes can modify the value of a variable in the class directly.)
If the transition action is an *event* (i.e. a transition occurs when something happens) look for a behavior or set of class behaviors that achieve that event.

If the transition action is a *constraint* (i.e. a transition occurs when some expression becomes true or false) look for behaviors that can change the value of the constraint expression. For example, note the constraints “[payment ok]” and “[payment not ok]” in example 5. These describe when the actions they describe can happen, based on the status of payment.

Does the class encapsulate behaviors to deal with the modeled actions? If not, you have found a defect of inconsistency. The state diagram specifies certain actions that no class behavior is capable of implementing.
Could you identify the object data used to verify the constraints? Are they consistent between the state diagram and the class description? If not, you have found an inconsistency. The state diagram describes certain constraints that must be checked but the class description as described may not support this check.

3) **Compare the class diagram to the state diagram to make sure that the class, as described, can capture the appropriate functionality.**

INPUTS: Object States (marked in blue on SD);
Transition Actions (marked in green on SD).

OUTPUTS: Discrepancy reports.

☞ Consider the system functionality in which this class participates, as described by the class description, and the states in which it may exist, as described by the state diagram.

Using your semantic knowledge of this class and the behaviors it should encapsulate, are all states described? If not, you have uncovered a defect of incorrect fact, that is, the class as described cannot behave as it should.

☞ Review the state diagram, looking for any unmarked states.

Is there some unstarred state? Could you evaluate the importance of this state? Does it really describe an essential object state? Is the state feasible considering all actions and constraints surrounding it? If yes, probably something is missing on the class diagram and there is an inconsistency between the diagrams. Otherwise, an extraneous fact should be reported.

☞ Review the state diagram, looking for any unmarked actions.

Is there some unstarred event? If yes, fill in a defect record showing the inconsistency between the class description and state diagram.

Is there some unstarred constraint? Is the constraint directly concerned with some object data? If yes, fill in a defect record showing the information that has been omitted from the class description.

B.3 Reading 3 -- Sequence x State diagrams

Goal: To verify that every state transition for an object can be achieved by the messages sent and received by that object.

Inputs to Process:

1. Sequence diagrams that describe the classes, objects, and possibly actors of a system and how they collaborate to capture services of the system.
2. State diagrams that describe the internal states in which an object may exist, and the possible transitions between states.

For **each state diagram,** perform the following steps:

1) **Read the state diagram to understand the possible states of the object and the actions that trigger transitions between them.**

INPUTS:	State diagram (SD).
OUTPUTS:	Transition Actions (marked and labeled in green on SD); Discrepancy reports.

☞ Determine which class is being modeled by this state diagram.

Could you identify the type of the object that this state machine is modeling? If you can't find this information, you have found a defect of omission. Fill in a discrepancy report for this.

☞ Trace the sequence of states and the *transition actions* (system changes during the lifetime of the object, which trigger a transition from one state to another) through the state diagram. Begin at the start state and follow the transitions until you reach the end state. Make sure you have covered all transitions.

☞ Highlight transition actions (represented by arrows) as you come to them using a green pen. For example, the state diagram provided in Example 5 contains seven transition actions. The arrow leading from the state labeled "authorizing" back to itself represents an action that does not actually change the state of the object. Give each action a unique label [A1, A2, …].

☞ Think about the states and actions you have just identified, and how they fit together. **Is it possible to understand and describe what is going on with the object just by reading this state machine? If not, you may have discovered a defect of ambiguity; the behavior of this class over its lifetime is not well described.**

2) **Read the sequence diagrams to understand how the transition actions are achieved by messages that are sent and received by the relevant object.**

INPUTS:	State diagram (SD); Transition Actions (marked and labeled in green on SD); Sequence diagrams (SqD).
OUTPUTS:	Object messages (marked and labeled in green on SqD); Discrepancy reports.

☞ Take the sequence diagrams and choose the ones that use the object modeled by the state diagram; use only this subset of the sequence diagrams in the remainder of this step.

Could you find sequence diagrams in which the object participates? If not, you have found a defect of omission (necessary descriptions of how this object contributes to system services are missing) or extraneous information (this object does not contribute to system services and is not needed). Fill out a discrepancy report describing this defect.

☞ For each sequence diagram identified in the previous step:

☞ Read the diagram to identify the system service being described and the messages that this object receives.

- ☞ Think about which object states on the state diagram are *semantically* related to the system service. Highlight the state transitions leading to and from these states, and use this subset for the remainder of this step.

- ☞ Map the object messages on the sequence diagram to the state transitions on the state diagram. Each transition action may map to one message, or a sequence of messages. To do this, you will need to think about the *semantics* behind the system messages. Are they contributing to achieving some larger system service or functionality? Do they have something to do with the types of states this object should be in? When you have made a mapping, mark the related messages and transition actions with a star (*). Label the messages with the same label given to their associated action on the state diagram. **Semantically, could you do this mapping? Were there additional messages needed to achieve the state transition? If so, you have discovered a defect of omission. Important messages are missing from the sequence diagrams for this object. Fill out a discrepancy report describing this defect.**

- ☞ Look for constraints and conditions on the messages you just mapped to state transitions. An example constraint might be "t>0", that is, whether or not a message is sent depends on the value of some attribute t. Look to see that any constraints/conditions found are captured somehow on the state diagram. This information might be captured by: 1) state information (i.e. the fact that t>0 corresponds to a particular state of the system; 2) transition information (i.e. some state transition occurs when t>0 becomes true or false; 3) nothing (i.e. this information is not relevant or important for the state diagram).

  Could you find some correspondence between the constraint/condition information on the state and sequence diagrams? Does the information included on both diagrams have the same meaning? If not, you have detected an inconsistency between the diagrams. Fill out a discrepancy report describing this defect. Is there some constraint/condition information on the sequence diagram that was not captured on the state diagram? If so, is it important enough to the idea of object states that it should have been somehow represented? If yes, you have detected an omission. Necessary information has been left out of the state diagram. Fill out a discrepancy report describing this defect.

3) Review the marked-up diagrams to make sure that all transition actions are accounted for.

INPUTS: Transition Actions (marked and labeled in green on SD);
Object messages (marked and labeled in green on SqD);.

OUTPUTS: Discrepancy reports.

- ☞ Review the state diagram looking for unstarred transition actions that could not be associated with object messages.

  If the transition action was labeled with a constraint, could you find a message or sequence of messages capable of satisfying the constraint? If not, you have found a defect of inconsistency. The state diagram requires system services that are not described on any sequence diagram. Fill out a discrepancy report describing this defect. If the transition action was labeled with an event, could you find a message, a sequence of messages, or some event performed by an actor that achieves the transition action? If not, you have found a defect of inconsistency. The state diagram requires system services that are not described on any sequence diagram. Fill out a discrepancy report describing this defect.

- ☞ If the starred messages and transition actions identified in the previous step appear on the same sequence diagram, make sure they appear in a logical order. That is, suppose the messages that achieve action A1 appear before the messages that achieve action A2 on one sequence diagram. This means that A1 must take place chronologically before A2. Then you should make sure that A1 could be reached before A2 on the state diagram as well.

  Does the order of events not match between the two diagrams? If so, there is an inconsistency. The system services and functionality are described in different ways on the two diagrams. Fill out a discrepancy report describing this defect.

B.4 Reading 4 -- Class diagrams x Class descriptions

Goal: To verify that the detailed descriptions of classes contain all the information necessary according to the class diagram, and that the description of classes make semantic sense.

Inputs to Process:

1. A class diagram (possibly divided into packages) that describes the classes of a system and how they are associated.

3. A set of class descriptions that lists the classes of a system along with their attributes and behaviors.

1) Read the class diagram to understand the necessary properties of the classes in the system.

INPUTS: Class diagram;
Class description.

OUTPUTS: Discrepancy reports.

For **each class on the class diagram,** perform the following steps:

☞ Find the relevant class description. Mark the class description with a blue symbol (*) when found. **If you can't find the description, you have found a defect of omission. This class was represented but not described on the class diagram. Fill in a discrepancy report for this.**

☞ Check the name and textual description of the class to ensure that they provide a *meaningful* description of the class that you are considering at this time. Also check that the description is using an adequate abstraction level.

Can you understand the purpose of this class from the high-level description? If not, the description may be too ambiguous to be used for the design model. Fill out a discrepancy report describing the situation.

☞ Verify that all the attributes are described along with basic types.

Are the same set of attributes present in both the class description and the class diagram? If not, it means that attributes are not appropriately described. This represents an inconsistency between the documents. Fill in a discrepancy report describing this situation and showing which document didn't capture the appropriate information.

Can this class *meaningfully* encapsulate all these attributes? That is, does it make sense to have these attributes in the class description? Are the basic types assigned to the attributes *feasible* according to the description of the attribute? If not, it represents an ambiguity or an incorrect fact. Fill in a discrepancy report describing this situation and showing which document didn't capture the appropriate information.

☞ Verify that all the behaviors and constraints are described.

Are the same set of behaviors and constraints present in both the class description and the class diagram? Does the class description use the same style or level of granularity (e.g. pseudocode) to describe all of the behaviors? If not, you have discovered an inconsistency. Different information is contained in different documents, or different styles are used within the same document. Fill in a discrepancy report describing this situation and showing which document didn't capture the appropriate information.

Can this class *meaningfully* encapsulate all these behaviors? Do the constraints make sense for this class? If not, it represents an ambiguity or an incorrect fact. Fill in a discrepancy report describing this situation.

Do the behaviors accomplish this procedure using attributes that have been defined (for this or some other class)? Are the constraints satisfiable using the attributes and behaviors that have been defined? If not, you may have discovered an omission or ambiguity. The behaviors and constraints as defined cannot be satisfied using the attributes and behaviors that have been defined. It may be that

new attributes must be included in the design, or the definition of the constraint/behavior changed. Fill in a discrepancy report describing the problem.

Do the behaviors for this class rely *excessively* on the attributes of other classes to accomplish their functionality? (Note that you must make a value judgement about what is meant by "excessive reliance." You should compare the number of references to other classes for this class with the rest of the system, and consider the type of functionality addressed to determine if such reliance is really necessary.) If so, you have uncovered a potential style issue: unnecessarily high coupling. Fill in a discrepancy report describing the problem as a "miscellaneous" defect.

☞ If the class diagram specifies any inheritance mechanisms for this class**,** verify that they are correctly described.
Is the inheritance relationship included on the class description? If not, you have uncovered a defect of omitted information. The class diagram specifies that this class is part of an inheritance hierarchy that should be described in the class description. Fill in a discrepancy report describing this situation. Use the class hierarchy to find the parents of this class. Is it true that, *semantically,* a <class name> is a type of <parent name>? Does it make sense to have this class at this point of the hierarchy? If not, you have uncovered a potential style issue: the hierarchy should not be defined in this way. Fill in a discrepancy report describing the problem as a "miscellaneous" defect.

☞ Verify that all the class relationships (association, aggregation and composition) are correctly described with respect to multiplicity indications.

Were the object roles captured on the class description? Is the correct graphical notation used on the class diagram to denote the type of relationship? If not, you have discovered an inconsistency; the information on the two diagrams does not agree. Fill in a discrepancy report describing this situation and showing which document didn't capture the appropriate information. *Semantically*, do the relationships make sense given the role and the objects related? For example, if a composition relationship is involved, do the connected objects really seem like a "whole-part" relationship? If so, you have uncovered a potential style issue: the relationships should not be defined in this way. Fill in a discrepancy report describing the problem as a "miscellaneous" defect. If cardinalities are important, were they described in the class description? Given your understanding of the relationship, do the quantities of objects used seem enough? If not, you may have discovered an inconsistency. Fill in a discrepancy report describing the problem. Is there some attribute representing the relationship? Does it use a feasible basic type, or structure of basic types (if multiple cardinality is involved)? If not, you may have discovered an inconsistency (if the documents do not agree). Fill in a discrepancy report describing the problem.

2) Review the class descriptions for extraneous information.

INPUTS: Class description.

OUTPUTS: Discrepancy reports.

☞ Review the class descriptions to make sure that all classes described actually appear in the class diagram. **Is there an unstarred class description? If so, you have discovered a defect of extraneous information. Class information has been described that does not actually participate in the class diagram. Fill in a discrepancy report describing the problem.**

B.5 Reading 5 -- Class Descriptions x Requirements Description

Goal: To verify that the concepts and services that are described by the functional requirements are captured appropriately by the class descriptions.

Inputs to Process:

1. A set of functional requirements that describes the concepts and services that are necessary in the final system.
2. A set of class descriptions that lists the classes of a system along with their attributes and behaviors.

1) **Read the requirements description to understand the functionality described.**

INPUTS: Set of functional requirements (FR).

OUTPUTS: Candidate classes/objects/attributes (marked in blue in FRs);
Candidate services (marked in green in FRs);
Constraints or conditions on services (marked in yellow in FRs).

- ☞ Read over the each functional requirement to understand the functionality that it describes.
- ☞ Find the nouns in the requirement; they are candidates to become classes, objects, or attributes in the system design. Underline the nouns with a blue pen.
- ☞ Find the verbs, or descriptions of actions, which are candidates to be services or behaviors in the system. Underline the verbs or action descriptions with a green pen.
- ☞ Look for descriptions of constraints or conditions on the nouns and verbs you identified in the preceding two steps. Especially pay attention to non-functional requirements, which typically contain restrictions and conditions on system functionality. For example, examine whether relationships between the concepts have been identified. Ask whether there are explicit constraints or limitations on the way actions are performed. Try to notice if definite quantities have been specified at any point in the requirement (see Example 4). Underline these conditions and constraints with a yellow pen.

2) **Compare the class descriptions to the requirements to verify if the requirements were captured appropriately.**

INPUTS: Set of functional requirements (FR);
Class description (CD).

OUTPUTS: Corresponding concepts have been marked on the FR and CD;
Discrepancy reports.

- ☞ For each green-underlined action description in the functional requirements, try to find an associated behavior or combination of behaviors in the class description. Use syntactic clues (e.g. a behavior name that is similar or synonymous to an action description) to help your search, but make sure the *semantic* meaning of the function in the requirements and high-level design is the same. When found, mark both the name of the behavior(s) in the class description and the description of the activity in the requirements with a green symbol (*).

  Do the classes receive the right information for accomplishing the required behaviors? Are *feasible* results produced? If not, you have found an incorrect fact. The classes as defined cannot achieve an appropriate service. Fill out a discrepancy report describing the problem.

- ☞ For each blue-underlined noun in the functional requirements, try to find an associated class in the class description. An associated class may be named after a concept from the requirements, may describe a general class of which the concept is a particular instance (i.e. an object), or may contain the concept as an attribute. Use syntactic clues (e.g. a class name that is similar to the name of a concept) to help your search, but make sure the *semantic* meaning of the concepts in the requirements and design is the same.
- ☞ If the concept in the functional requirements corresponds to a class name in the class description, mark both the name of the class in the class description and the concept in the requirements description with a blue symbol (*).

Do the class descriptions contain sufficient information regarding the concepts that play some role in this functionality? Do the class names have some connection to the nouns you had marked? Are the classes using unambiguous and clear information to describe the concepts? If not, you have detected an ambiguity. Fill out a discrepancy report describing the problem.

Do these classes encapsulate (blue-marked) attributes concerned with the nouns you had marked? Do these classes encapsulate (green-marked) behaviors concerned with the verbs or actions descriptions you had marked? Were all identified constraints and conditions for these classes regarding this requirement described? If not, you have found an omission; important information from the requirements has been left out. Fill out a discrepancy report describing the problem.

☞ If the concept in the functional requirements corresponds to an attribute in the class description, mark both the name of the attribute in the class description and the concept in the requirements description with a blue symbol (*).

Is the class description using *feasible* types to represent information, given the requirements description? Were the (yellow-underlined) constraints and conditions on these attributes observed in their definition? If not, you have found an incorrect fact. Fill out a discrepancy report describing the problem.

3) **Review the class description and functional requirements to make sure that all appropriate concepts correspond between the documents.**

INPUTS: Set of functional requirements (FR);
Class description (CD).

OUTPUTS: Discrepancy reports.

☞ Look for descriptions of functionality in the requirements that have been omitted from the design. **Is there some underlined concept (in blue) or activity (in green) in the requirements, which is unstarred? If yes, it may mean that some concept was not captured in the design. However, it may also mean that some concept in the requirements was simply used for explanation or example, and need not be made a part of the system. Decide whether this omission should be identified as a defect. Describe what is missing, filling in a defect record for each unstarred noun.**

B.6 Reading 6 -- Sequence Diagrams x Use-cases

Goal: To verify that sequence diagrams describe an appropriate combination of objects and messages that work to capture the functionality described by the use case.

Inputs to process:

1. A use case that describes important concepts of the system (which may eventually be represented as objects, classes, or attributes) and the services it provides.
2. One or more sequence diagrams that describe the objects of a system and the services it provides. There may be multiple sequence diagrams for a given use case since a use case will typically describe multiple "execution paths" through the system functionality. The correct set of sequence diagrams for a use case must be selected by using traceability information, or by someone with semantic knowledge about the system. Finding the correct set of sequence diagrams without traceability information or knowledge of the system will be hard.
3. The class descriptions of all classes in the sequence diagram.

1) **Identify the functionality described by a use case, and important concepts of the system that are necessary to achieve that functionality.**

INPUTS: Use case (UC)

OUTPUTS: System concepts (marked in blue on UC);
Services provided by system (marked in green on UC);
Data necessary for achieving services (marked in yellow on UC).

☞ Read over the use case to understand the functionality that it describes.

☞ Find the nouns included in the use case; they describe concepts of the system. Underline and number each unique noun with a blue pen as it is found. (That is, if a particular noun appears several times, label the noun with the same number each time.)

☞ For each noun identify the verbs that describe actions applied *to* or *by* the nouns. Underline the identified services and number them (in the order they must be performed) with a green pen. Look for the constraints and conditions that are necessary in order for this set of actions to be performed. As an example, consider Example 1, in which constraints and conditions have been highlighted. In this use case, there is an example of both a constraint ("The Customer can only wait for 30 seconds for the authorization process") and a condition ("time of payment is the same as the purchase time").

☞ Also identify any information or data that is required to be sent or received in order to perform the actions. Label the data in yellow as "Di,j" where subscripts i and j are the numbers given to the nouns between which the information is exchanged.

2) **Identify and inspect the related sequence diagrams, to identify if the corresponding functionality is described accurately and whether behaviors and data are represented in the right order.**

INPUTS: Use case, with concepts, services, and data marked;
Sequence diagram (SD).

OUTPUTS: System concepts (marked in blue on SD);
Services provided by system (marked in green on SD);
Data exchanged between objects (marked in yellow on SD).

☞ For each sequence diagram, underline the system objects with a blue pen. Number them with the corresponding number from the use case.

☞ Identify the *services* described by the sequence diagrams. To do this, you will need to examine the information exchanged between objects and classes on the sequence diagrams (the horizontal arrows). If the information exchanged is very detailed, at the level of messages, you may need to abstract several messages together to understand the services they work to provide. Underline the identified services and number them (in the order they occur in the diagram) with a green pen. Look for the condition that activates the actions.

- ☞ Identify the information (or data) that is exchanged between system classes. Label the data in yellow as "Di,j" where subscripts i and j are the numbers given to the objects between which the information is exchanged.

3) **Compare the marked-up diagrams to determine whether they represent the same domain concepts.**

INPUTS: Use case, with concepts, services, and data marked;
Sequence diagram, with objects, services, and data marked.

OUTPUTS: Discrepancy reports.

- ☞ For each of the blue-marked nouns on the use case, search the sequence diagram to see if the same noun is represented. Mark the noun on the use case with a blue star (*) if it can be found on the sequence diagram. **Is there an unstarred noun on the use case? If yes, it means that a concept was used to describe functionality on the use case but was not represented on the sequence diagram. For each of the nouns on the sequence diagram, find the corresponding class on the class description and check whether the unstarred noun is an attribute. If the unstarred noun does not appear as an attribute of any of these classes, you have found an omission. A concept was described on the use case but has not appeared in the system design. Fill in a discrepancy report to describe the problem.**

- ☞ Mark the nouns on the sequence diagram with a blue star (*) if they appear only on the sequence diagram. **Is there a starred** (*) **noun on the sequence diagram? If so, you have found an extraneous noun, or a noun describing a lower-level concept, on the sequence diagram. Think about whether the concept is necessary for the high-level design, and whether it represents a level of detail that is appropriate at this time. If necessary, fill in a discrepancy report describing the problem.**

- ☞ Identify the services described by the sequence diagram, and compare them with the description used on the use case. Are the classes/objects exchanging messages in the same order specified on the use case? Were the data that appear on messages on the sequence diagram correctly described on the use case? Is it possible for you to understand the expected functionality just by reading the sequence diagram? **Do the classes exchange messages in the same specified order? If not think about whether this represents a defect. Usually, switching the order of messages may have an effect on the functionality. But sometimes messages can be switched without affecting the outcome; other times, messages can be performed in parallel, or conditions may ensure that only one or the other message is executed anyway. If a defect has been found, fill in a discrepancy report describing the problem. Are all transported data in the right message? Is data being sent between the right two classes (i.e. do the labels "Di,j" for the data match between diagrams)? Do the messages make sense for the objects sending and receiving them, and do they make sense for achieving the relevant services? If not, it means that the sequence diagram is using information incorrectly. Fill in a discrepancy report describing the problem.**

- ☞ Are all the constraints and conditions from the use case being observed in this sequence diagram? Is some detail from the use case missing here?

  Were the constraints observed? Was all of the behavior and data on the sequence diagram directly concerned with the use-case? If no, it means that the sequence diagram is using information incorrectly. Fill in a discrepancy report on this matter.

B.7 Reading 7 -- State Diagrams x Requirements Description and Use-cases

Goal: To verify that the state diagrams describe appropriate states of objects and events that trigger state changes as described by the requirements and use cases.

Inputs to process:

1. The set of all state diagrams, each of which describes an object in the system.
2. A set of functional requirements that describes the concepts and services that are necessary in the final system.
3. The set of use cases that describe the important concepts of the system

For each state diagram, do the following steps:

1) **Read the state diagram to basically understand the object it is modeling.**

2) **Read the requirements description to determine the possible states of the object, which states are adjacent to each other, and events that cause the state changes.**

   INPUTS: Requirements Description (RD)

   OUTPUTS: Object States (marked in blue on SD)

   Adjacency Matrix

   - ☞ Put away the state diagram and erase any (*) from that are in the requirements from previous iterations of this step. Now, read through the requirements looking for places where the concept is described or for any functional requirements in which the concept participates or is affected. When you locate one of these, mark it in pencil with a (*) so that it will be easier to use for the remainder of the step. Focus on these parts of the RD for the rest of the step.
   - ☞ Locate descriptions of all of the different states that this object can be in. To locate a state, look for attribute vales or combinations of attribute values that can cause the object to behave in a different way. When you locate a state underline it with a blue pen and give it a number.
   - ☞ Now identify which one of the numbered states is the Initial state. Using a blue pen, mark it with an “I”. Likewise mark the end state with an “E”.
   - ☞ When you have found all of the states, on a separate sheet of paper, create a matrix with 1..N across the top and 1..N down the left side, where 1..N represents the numbers that you gave to the states in the previous step.
   - ☞ For each pair of states, if the object can change from the state represented by the number on the left hand side to the state represented by the number on the top row, then mark the box at the intersection of the row and column. If you can determine the event(s) that cause the state change put that in the box, if not just put a check mark (the event will be determined in a later step). If you can determine that it is not possible for the transition to happen then place an X in the box. If you cannot make a definite determination then leave the box blank for now.
   - ☞ For any event that you have identified above, if there are any constraints described in the requirements, then write those by the event in the matrix.

3) **Read the Use cases and determine the events that can cause state changes.**

   INPUT: Use Cases

   OUTPUT: Completed Adjacency Matrix

   - ☞ Read through the use cases and find the ones in which the object participates. Focus on these for the rest of the step.

- ☞ For each box in the adjacency matrix that has a check mark in it, look through the use cases and determine what event(s) can cause that transition. These events may not be obvious and may require you to abstract the use-cases and think about what is actually going on with each object. Erase the check mark and write this event(s) in its place.
- ☞ For each box that is blank in the adjacency matrix, see if any event that can cause that transition is described in the use cases. If it is, then write that event in the box, if not then place an X in the box.

4) **Read the state diagram to determine if the states described are consistent with the requirements and if the transitions are consistent with the requirements and use cases.**

INPUT: Requirements Description;

State Diagram (SD);

Adjacency Matrix (AM).

OUTPUT: Discrepancy Reports

- ☞ For each state that is marked and numbered in the requirements description, find the corresponding state on the state diagram and using a blue pen, mark it with the same number used in the requirements. Be careful, because the same state may have a different name in the requirements than it has on the state diagram. To determine if two different names are talking about the same state, you must use your understanding of the requirement's description of the state and the information contained in the state diagram. This may be an iterative process where if states appear to be missing, you must go back and look again at what you have identified and make sure that it is correct.

  Were you able to find all of the states?

  If a state is missing, look to see if two or more states that you marked in the requirements were combined into one state on the state diagram. If not, then you have found a defect of Omission. If so, then does this combination make sense? If not, you have found a defect of Incorrect Fact.

  Where there extra states in the state diagram?

  Look to see if one state that you marked in the requirements has been split into two or more states in the state diagram. If not, then you have found a defect of Extraneous. If so, does this split make sense? If not, you have found a defect of Incorrect Fact.

- ☞ Once you have all of the states labeled with numbers, using the AM, compare the transition events marked on the matrix to the ones on the SD. For any box on the AM that is marked with an event, check the corresponding states on the SD to make sure they have an event to transition between them, and check to ensure that the event is the same.

  Do all of the events on the AM appear on the SD? If not, you have found a defect of omission. Do events appear on the SD that are not on the AM? If so, you have found a defect of extraneous fact.

- ☞ For each constraint that was marked on the AM, find it on the SD.

  Did you find all of the constraints that are on the AM? If not, then you have found a defect of omission. Did you find a constraint on the state diagram that is not on the AM? If so, does the constraint make sense? If not then you have found a defect of extraneous fact.

APPENDIX C - The Defect report forms

C.1 - Sequence x Class Diagrams Discrepancy Report

Fill in one record for each defect

Document: Sequence Diagram

Concept Name:
Granularity: (actor, object, message, data, condition)
Type of defect: (inconsistency, ambiguity, incorrect fact, extraneous, miscellaneous)
Description:

Document: Class Diagram

Which classes are (or should be) involved?
Concept Name:
Granularity: (attributes, behavior, condition, inheritance, relationship, cardinalities, roles)
Type of defect: (inconsistency, ambiguity, incorrect fact, extraneous, miscellaneous)
Description:

C.2 - State Diagram x Class Description Discrepancy Report

Fill in one record for each defect

Document: State Diagram

Object/Class Name:
Granularity: (state, event, behavior, condition)
Concept Name:
Type of defect: (inconsistency, extraneous, miscellaneous)
Description:

Document: Class Description

Class Name:
Granularity: (attribute, behavior, condition)
Concept Name:
Type of defect: (inconsistency, extraneous, miscellaneous)
Description:

C.3 - State x Sequence Diagrams Defects Report

Fill in one record for each defect

Document: State Diagram

Object/Class Name:
Granularity: (state, event, behavior, condition)
Concept Name:
Type of defect: (inconsistency, extraneous, miscellaneous)
Description:

Document: Sequence Diagram

Class Name:
Granularity: (behavior, condition)
Concept Name:
Type of defect: (inconsistency, extraneous, miscellaneous)
Description:

C.4 - Class Diagram x Class Description Discrepancy Report

Fill in one record for each defect

Document: Class Descriptions

Class Name:
Granularity: (attributes, behavior, condition, inheritance, relationship)
Concept Name:

Type of defect: (inconsistency, ambiguity, incorrect fact, extraneous, miscellaneous)
Description:

Document: Class Diagram
Class Name:
Granularity: (attributes, behavior, condition, inheritance, relationship, cardinalities, roles)
Concept Name:
Type of defect: (inconsistency, ambiguity, incorrect fact, extraneous, miscellaneous)
Description:

C.5 - Class Description x Requirement Description Discrepancy Report
Fill in one record for each defect

Document: Class Descriptions
Class Name:
Granularity: (class/objects, attributes, behavior, condition, inheritance, relationship)
Concept name:
Type of defect: (omission, ambiguity, incorrect fact, extraneous, miscellaneous)
Description:

C.6 - Sequence Diagrams x Use Cases Discrepancy Report
Fill in one record for each defect

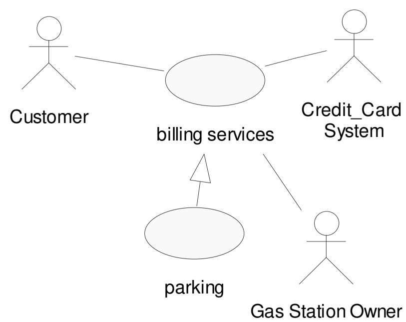
Document: Sequence Diagrams
Which classes are (or should be) involved?
Concept name:
Granularity: (class/objects, data, service, condition)
Type of defect: (omission, ambiguity, incorrect fact, extraneous, miscellaneous)
Description:

C.7 - State Diagrams x Requirements Description Report
Fill in one record for each defect

Document: State Diagram
Which classes are (or should be) involved?
Concept name:
Granularity: (state, event, behavior, condition)
Type of defect: (inconsistency, extraneous, miscellaneous)
Description:

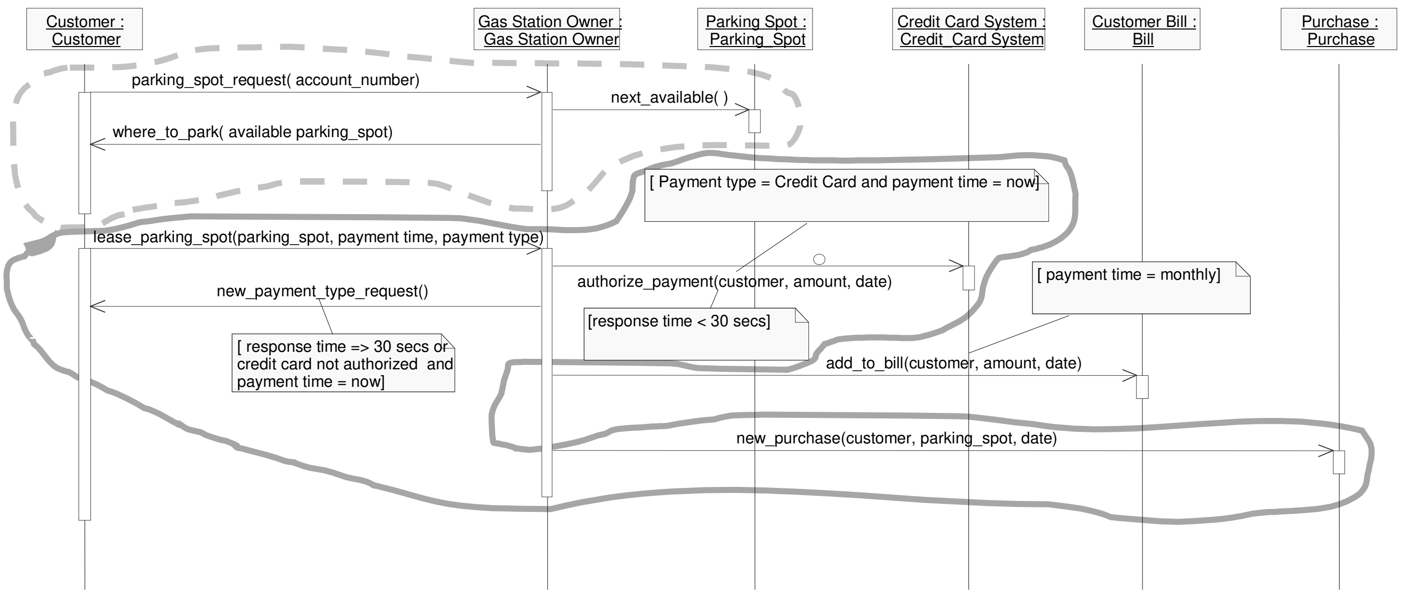
APPENDIX D – The Examples of Documents to Be Read

Example 1: A use case for an automated system at a gas station, describing how a customer purchases a parking spot. Note that “time of payment is the same as purchase time” is a *condition*; it describes what must be true for the functionality to be executed. “The Customer can only wait for 30 seconds for the authorization process” imposes a *constraint* that must be always be true for system functionality.

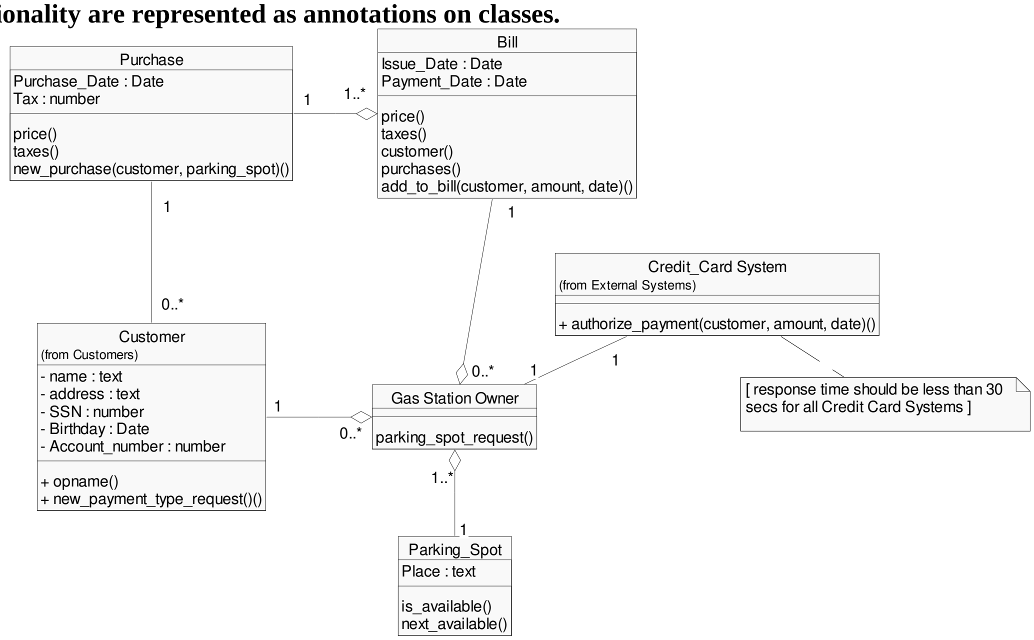


A customer, giving his account_number, asks the Gas Station Owner for an available parking spot to park his car.
To get an available parking spot Gas Station Owner searches for the next parking place available.
With this information the customer can confirm the lease of the parking place. The time of payment (time of purchase or a monthly paper bill) and how the service should be paid (by cash, personal check or credit card).
If the time of payment is the same as the purchase time and Customer decides to pay by Credit Card then Credit Card system should be used. The Customer can only wait for 30 seconds for the authorization process otherwise this payment should be made by cash or personal check to avoid other Customers waiting on the lane. The Gas Station Owner should ask the Customer for a new payment type.
It allows the Gas Station Owner to mark a new service purchase for this Customer at this date.

Example 2: A sequence diagram for the automated gas station system, capturing how classes collaborate to perform the functionality described in Example 1. Combinations of messages that form system *services* have been marked. Conditions and constraints are included as annotations on the diagram. "Response time < 30 secs" represents a nonfunctional *constraint* on the way certain functionality has to be implemented. "Payment time = monthly" is an example of a *condition* that must be true for a particular message to be executed; in this case, the system variable "payment time" must have the value "monthly."



Example 3: The class diagram for the classes described in Example 2. Note that *constraints* on system functionality are represented as annotations on classes.



Example 4: Requirements descriptions and Class descriptions used to show how conditions and constraints should be considered while reading both documents. Observe the relationship between both documents shown by the underlined information.

Requirement Description

1 – A customer has the option to be billed automatically at the time of purchase (of gas, car maintenance or parking spots) or to be sent a monthly paper bill. Customers can pay via cash, credit card or personal check. Gas Station services have a fixed price (gas: US$ 1.09 gallon, <u>car maintenance: US$ 150.00</u> and parking spot: US$ 5.00 per day). <u>The tax is 5% added to the final price of the purchase</u>. Sometimes, the <u>Gas Station owner can define discounts to those prices</u>.

Class Description

Class name: Purchase
Category: Customers
External Documents:
Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: none
Public Interface:
Operations:
price
taxes
Private Interface:
Attributes:
Purchase_Date : Date
Tax : number
Service: Services
Implementation:
Attributes:
Purchase_Date : Date
<u>Tax : number = 0.05</u>
Operation name: price
Public member of: Purchase
Concurrency: Sequential
<u>Return (1 + tax) * service->price</u>
Operation name: taxes
Public member of: Purchase
Concurrency: Sequential
Return tax * service->price

Class name: Services
Category: Services
External Documents:
Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: none
Public Interface:
Operations:
price
Private Interface:
Attributes:
Discount_Rate : number
Price : number
Implementation:
Attributes:
Discount_Rate : number
Price: number
Operation name: price
Public member of: Services
Concurrency: Sequential
<u>Return (1 - discount_rate) * price</u>

Class name: Car_Maintenance
Category: Services
External Documents:
Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Services
Public Interface:
Operations:
price
Private Interface:
Attributes:
Price : number
Implementation:
Attributes:
<u>Price : number = 150.00</u>

Example 5: A state diagram for the “gas station owner” class from the automated gas station system. An associated sequence diagram is shown in Example 2.