Technical Report

Reading Techniques for OO Design Inspections

Guilherme H. Travassos [1] ght@cos.ufrj.br
Forrest Shull [2] fshull@fc-md.umd.edu
Jeffrey Carver [3] carver@cs.umd.edu
Victor Basili [2,3] basili@cs.umd.edu

[1] Systems Engineering and Computer Science Department
COPPE
Federal University of Rio de Janeiro
C.P. 68511 - Ilha do Fundão
Rio de Janeiro, RJ 21945-180
Brazil

[2] Fraunhofer Center - Maryland
University of Maryland
4321 Hartwick Road
Suite 500
College Park, MD 20742
USA

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

ABSTRACT
Inspections can be used to identify defects in software artifacts. In this way, inspection methods help to improve software quality, especially when used early in software development. Inspections of software design can be especially crucial since design defects (problems of correctness and completeness with respect to the requirements, internal consistency, or other quality attributes) can directly affect the quality of, and effort required for, the implementation. We have created a new family of "reading techniques" (so called because they help a reviewer to "read" a design artifact for the purpose of finding relevant information) that gives specific and practical guidance for identifying defects in Object-Oriented designs. Each reading technique in the family focuses the reviewer on some aspect of the design, with the goal that an inspection team applying the entire family should achieve a high degree of coverage of the design defects. In this paper, we present an overview of this new set of reading techniques. We discuss how these techniques were developed and suggest how readers can use them to detect defects in high level object oriented design UML diagrams.

General Terms
Measurement, Design, Experimentation, Verification

Keywords
Empirical studies, OO design inspections, software process, experimental process, software quality

1. INTRODUCTION

A software inspection aims to guarantee that a particular software artifact is complete, consistent, unambiguous, and contains as less defects as possible to effectively support further system development. For instance, inspections have been used to improve the quality of a system's design and code [5].

Because they rely on human understanding to detect defects, inspections have the advantage that they can be performed as soon as a software work artifact is written and can be used with of different artifacts and notations. Typically, inspections require individuals to review a particular artifact, and then meet as a team to discuss and record defects, which are then sent to the document's author to be corrected. Because inspections are typically performed by teams, they are a useful way of sharing technical expertise about the quality of the software artifacts among the participants. And, because developers become familiar with the idea of reading each other's artifacts, they can lead to more readable artifacts being produced over time.

On the other hand, the dependence on human effort causes non-technical issues to become a factor: reviewers can have different levels of relevant expertise, can get bored if asked to review large artifacts, can have their own feelings about what is or is not important, or can be affected by political or personal issues. For this reason, there has been an emphasis on defining processes that people can use for performing effective inspections.

Most publications concerning software inspections have concentrated on improving the inspection meetings while assuming that individual reviewers are able to effectively detect defects in software documents on their own. Fagan [6] and Gilb and Graham [9] emphasize the inspection *method*[1], identifying multiple phases involving planning, defect detection, defect collection, and correction. As the basis for many of the review processes now in place (e.g., at NASA [12]), they have inspired the direction of much of the research in this area, which has tended to concentrate on improving the review *method*. However, they do not give any guidelines to the reviewer as to how defects should be found in the detection phase; both assume that the individual review of these documents can already be done effectively.

Proposed improvements to Fagan's method often center on the importance and cost of the meeting. However, empirical evidence has questioned the importance of team meetings by showing that meetings do not contribute to finding a significant number of new defects that were not already found by individual reviewers [24][13]. This line of research suggests that efforts to improve the review *technique*, that is, the process used by each reviewer to find defects in the first place, could be of benefit.

One approach to doing this is provided by *software reading techniques*. A reading technique is a series of steps for the individual analysis of a software product to achieve the understanding needed for a particular task [2]. Reading techniques attempt to increase the effectiveness of inspections by providing procedural guidelines that can be used by individual reviewers to examine (or "read") a given software artifact and identify defects. Rather than leave reviewers to their own devices, reading techniques attempt to capture knowledge about best practices for defect detection into a procedure that can be followed. Families of reading techniques have been tailored to defect inspections of requirements (for requirements expressed in English or SCR, a formal notation) and to usability inspections of user interfaces. There is empirical evidence that software reading increases the effectiveness of inspections on different types of software artifacts, not just limited to source code [13][1][2][8][14][25].

In this work, we describe a family of software reading techniques for the purpose of defect detection of high-level Object-Oriented (OO) designs diagrams represented using Unified Modeling Language (UML) [7]. A high level design is a set of artifacts concerned with the representation of real world concepts. As a consequence of using the object-oriented paradigm these concepts are represented as a collection of discrete objects that incorporate both data structure and behavior. The Object-Oriented Reading Techniques (OORTs) consist of 7 different techniques that support the reading of different design diagrams. More specifically, the reading techniques described in this work are tailored to inspections of high-level design artifacts that capture the static and dynamic views of the problem using UML notation: class, sequence, and state diagrams. Usually, these are the main UML diagrams that developers build for high-level OO design. To compare design contents against requirements, we expect that there will be a textual description of the functional requirements that may also describe certain behaviors using more specialized representations such as use-cases [10].

The development of these techniques has been supported by a series of empirical studies [22], which addresses questions aiming to identify the feasibility, technical soundness, usability and applicability of such techniques. The results we have so far provide evidence that the OORTs are feasible and can support readers in identifying different types of design defects [19][16][20].

Section 2 briefly describes object-oriented design in terms of the information that is important to be checked during software inspections, showing an outline of the whole set of techniques and the different types of defects such techniques are intended to identify. Section 3 discusses how the techniques were developed and validated. Section 4 introduces the reading techniques and discusses

[1] In this text we distinguish a "technique" from a "method" as follows: A technique is a series of steps, at some level of detail, which can be followed in sequence to complete a particular task. We use the term "method" as defined in [1], "a management-level description of when and how to apply techniques, which explains not only how to apply a technique, but also under what conditions the technique's application is appropriate."

the process inspectors can use to apply them. Finally, some suggestions for future work are discussed in the conclusions.

2. READING TECHNIQUES FOR HIGH LEVEL DESIGN

Each reading technique can be thought of as a set of procedural guidelines that reviewers can follow, step-by-step, to examine a set of diagrams and detect defects. The types of defects on which our techniques are focused, as listed in Table 1, are based on earlier work with requirements inspections [17]. The defect taxonomy is important because it helps focus the kinds of questions reviewers should answer during an inspection.

Table 1 – Types of software defects, and their specific definitions for OO designs

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.

We defined one reading technique for each pair or group of diagrams that could usefully be compared against each other. For example, use cases needed to be compared to interaction dia grams to detect whether the functionality described by the use case was captured and all the concepts and expected behaviors regarding this functionality were represented. The full set of our reading techniques is defined as illustrated in Figure 2, which differentiates horizontal[2] (comparisons of documents within a single lifecycle phase) from vertical[3] (comparisons of documents between phases) reading.

While horizontal reading aims to identify whether all of the design artifacts are describing the same system, vertical reading tries to verify whether those design artifacts represent the right system, which is described by the requirements and use-cases. So, the goal is that when all the techniques are used together, then all the quality issues in the design are covered. If the development team is not using the full set of UML design artifacts, then the corresponding review techniques need not be applied, without impact to the design inspection process[4].

The horizontal techniques should be performed before the vertical techniques, however, a subset or reordering of the techniques may be chosen based on important attributes of the design to be reviewed. This is particularly interesting when developers are dealing with specialized application domains. For example, consider a system whose functionality is based mainly on its reaction to stimuli where state machine diagrams are common. In this situation, it could be beneficial to use the reading techniques that focus on state machine diagrams before using the reading techniques that

[2] Consistency among documents is the most important feature here.
[3] Traceability between the phases is the most important feature here.
[4] However, this situation is not true for the software process as a whole. Some artifacts are important, such as a class diagram if missing implies that the design didn't capture the static view of the problem.

focus on the other design diagrams. For conventional systems, such as database systems, the semantic model of the information and the flow of the transactions seem to be the important information. Therefore, a subset of the techniques could be picked that focus on this information. In this situation, first reading the class diagram against the sequence diagrams seems to be a good idea then continuing with the rest of the techniques.

Further description about the process of applying the reading techniques can be found in Section 4 and in more detail in [21] and [23]. Information about the techniques and a complete definition for all the terms and definitions used in the context of this paper can be found in [16], which is accessible via the web.

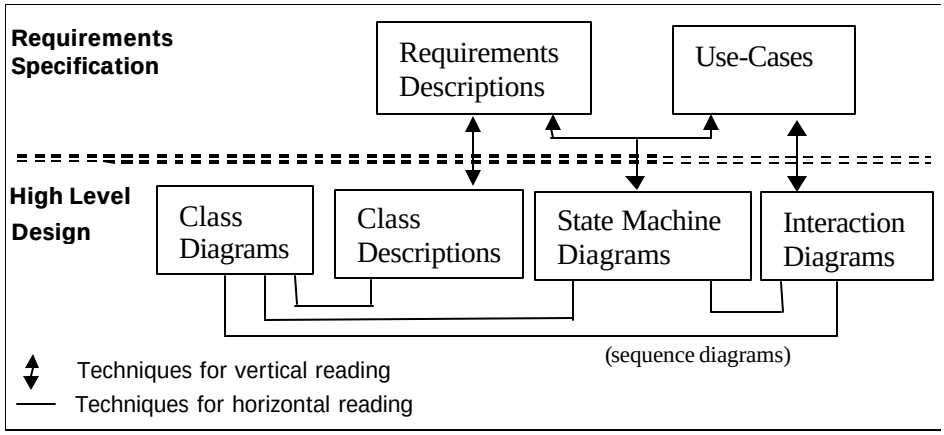


Figure 2 – Set of OO Reading Techniques

3. THE DEVELOPMENT OF OORTs

Since the first version, produced in 1998, the OO reading techniques have been modified and improved based on the results of a series of empirical studies. The sequence of studies and evolution of goals are illustrated in [18]; the results are summarized in the following sections. For the sake of clarity, we define the different types of studies run as follows:

Feasibility study: Data is collected according to some experimental design, but full control over all possible variables is not achieved. Such studies attempt to test the effectiveness of a process but are not able to rule out all rival hypotheses that may still exist at the end of the study. For example, we may observe changes in subject effectiveness but cannot completely rule out the possibility that they were caused by something other than the new process. The goal here is to provide the researcher with enough information to justify continued work.

Observational study: We use the term "observational" to define a setting in which an experimental subject performs some task while being observed by an experimenter. The purpose of the observation is to collect data about how the particular task is accomplished. Observational techniques can be used to understand current work practices that can be incorporated into the new process. They are also useful for getting a fine-grained understanding of how a new process is applied.

Case study: Case studies examine a particular process in the context of a larger software lifecycle. Case studies are usually not suitable vehicles for understanding a completely new process. They are expensive – subjects must be trained and must overcome the learning curve, and their time is potentially costly.

3.1 Evaluating Feasibility

Based on lessons learned from studying requirements inspections, and different types of OO design defects, an initial set of techniques was created. We chose to run a feasibility study, with feedback on form and content as a secondary goal, before expending effort to perfect the techniques. This initial validation was accomplished by means of a study [19] during the Fall 1998 semester at the University of Maryland (UMCP) that evaluated the feasibility of applying reading techniques to an OO design.

Subjects: The subjects came from a senior-level undergraduate software engineering course. Of the 44 students in the class, 32% had some previous industry experience in software design while 59% had classroom experience with design but had not used it on a real system (9% had no prior experience at all in software design). 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: An initial version of the reading techniques was 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 (11 classes in high-level design), but contained some design complexity due to non-functional performance requirements.

Procedure: Prior to this study, subjects performed an inspection of the requirements for the system, to detect defects and to better acquaint themselves with the given system and the domain. Subjects were then given the "corrected" requirements (based on the aggregate inspection results of the class) and use cases and asked to design the system. The "best" design, as chosen by course instructors, was distributed to the class and subjects were asked to perform a design inspection of it. Inspection activities consisted of an individual review followed by a team meeting (the main focus of which was to agree on a consensus list of defects). There was no control group. Therefore, we could not compare the OORTs' effectiveness to that of another OO inspection method. There were two reasons for this decision. The first was that we are aware of no other published methods for reading OO designs. Secondly, in a classroom environment, it was not possible to provide instruction on a topic to only a portion of the class. Each team applied all seven of the reading techniques, but divided them up to reduce workload: One member 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.

Data Collection: Questionnaires and interviews were used to collect qualitative data that addressed the question of feasibility directly, while analysis of artifacts was used as a control on the data quality and process conformance. Using both questionnaires and interviews allowed us to collect qualitative data at different times, under different conditions; evaluating the consistency of answers provided a first level check on data quality. The qualitative data concerned:

- Opinion of effectiveness of technique (measured by what percentage of the defects in the document subjects thought they had found)
- Subjective usefulness of different perspectives (open-ended question)
- How closely subjects followed the technique (collected qualitatively and quantitatively, for consistency)
- Practicality of the techniques; would subjects use some or all of them again (open-ended question)

The questionnaires were also used to capture limited quantitative data, namely the time required for individual review. Analysis of the subjects' defect lists yielded quantitative data concerning the number and type of defects detected by the techniques. (Because this was mainly a feasibility study,

we made the assumption in our counting of defects that all defects reported were real problems with the document.)

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).
- A majority of the subjects agreed that the techniques were helpful.
- Vertical techniques tended to find more defects of omitted and incorrect functionality.
- 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.

Thus, the data supported the conclusion that the techniques were feasible: they could really be used to detect defects, and moreover could be used to target particular types of defects.

At the same time, the qualitative data also indicated that the techniques were not as well specified as they could be. From the qualitative data we were able to learn three global lessons on how to improve the techniques for the second version:

- OO reading techniques should concentrate on semantic, not syntactic, issues.
- Reading techniques need to include not only instructions for the reader, but some motivation as to why those instructions are necessary.
- The level of granularity of the instructions needs to be precisely described. For instance, 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, and just assuming that subjects will intuitively grasp the correct level of granularity is naïve and causes frustration for the reviewer.

These results were at a global, non-specific level of detail. We found that the high level goal of the techniques, to find defects, was accomplished. In addion, global issues (that is, issues requiring changes to the general substance of the techniques rather than individual steps) about the process were uncovered such as including more semantic checking and better motivating the readers.

These results led us to produce a second version of the techniques that incorporated several global changes, such as a greater focus on semantic checking, more explanation of the goals of the process steps, and a new terminology to help discuss system functionality in more detail.

3.2 Observing the Technical Soundness

Because we were still interested in studying the techniques in isolation, rather than applied as part of a full software lifecycle, the new version was studied during the Fall 1999 semester at UMCP. The reason for this type of study was primarily that we wanted some indication about the problem domains, and the background of inspectors, for which the techniques could be most useful. The risk of introducing the techniques on an unsuitable project, with time and budget constraints, when their ease of use had not been tested also implied that another feasibility study could be useful.

To get the level of detail about the techniques that we wanted, we used an observational approach (i.e., using experimental methods suitable for understanding the process by which subjects apply the techniques) [16]. Because this observational approach was a somewhat unusual approach, we first performed a pilot study to debug the observational approach and get it to work in our setting. Only after that did we perform a full-scale observational study, reported below. The observational approach was necessary to understand what improvements might be necessary at the level of individual steps, for example, whether subjects experience difficulties or misunderstandings while

applying the technique (and how these problems may be corrected), whether each step of the technique contributes to achieving the overall goal, and whether the steps of the technique should be reordered to better correspond to subjects' own working styles.

Subjects: The 28 subjects were members of a graduate-level Software Engineering class. The subjects were grouped in pairs for this study, with one member of the team acting as the executor (responsible for applying the procedure) and the other as the observer (responsible for recording observations about how the procedure was executed). Of the 14 that actually performed the OO inspection, 86% had previous industry experience with OO design and the other 14% had classroom experience. All students received training on the OO reading techniques and the observation process.

Materials: The new version of the OO reading techniques was applied to two designs: one for the Loan Arranger (LA) system, described in the last section, and one for an automated parking garage control system (PGCS). The Loan Arranger design used in this study was a simpler version of the same system described in the feasibility study (7 classes in the high level design, 4 interaction diagrams and 3 state diagrams). The PGCS was responsible for allowing drivers to enter and leave a parking garage and keeping track of monthly parking tickets as well as the number of available spaces for general parking. The PGCS was a relatively small system (6 classes in the high level design, 5 interaction diagrams and 2 state diagrams). The LA problem domain was selected due to its unfamiliarity to reviewers, while the PGCS domain was familiar.

Procedure: A quasi-experimental, factorial design was used in which half of the class reviewed the LA design, and the other half the PGCS. Unlike the previous study, this experiment consisted of individual review only; inspectors did not meet as teams. In each of these groups, roughly half the subjects had previously inspected the requirements document for the same system. In this scheme, we could look for any differences in performance due to the reviewers' past familiarity with the system requirements or with the problem domain.

Before the study, subjects received training in the reading techniques to be applied and the observational methods. Training in observational methods was accomplished by presenting the roles of executor and observer and defining their specific responsibilities. Subjects were asked to come up with their own questions for eliciting information about the overall effectiveness of the techniques and the way in which the process was applied (e.g. if the procedure was too detailed or missing key information). After the execution of the techniques, each team wrote an evaluation report discussing their experience and the results of the observation.

Data Collection: Analysis of artifacts was again used for collecting some quantitative data, namely the time required for executing the techniques and the number and type of defects detected. However, observational techniques were the most important method used in this study. A rich array of qualitative data was collected through their use. As mentioned earlier, the teams produced an evaluation report, which included both a summary of the notes taken during observation as well as retrospective data determined after the execution of the process. Some of the metrics collected from the observations include:

- Executor's opinion of effectiveness of technique
- Problems encountered with specific steps of procedure
- How closely executors followed the techniques

The retrospective data (collected via open-ended questions) provided the following information:

- Usefulness of different perspectives
- Practicality of the techniques; would they use some or all of them again
- The problems encountered using the techniques

As can be noticed, the retrospective data are better suited to global issues, rather than the critiquing of individual steps. Also, some of the metrics collected here were the same as in the previous feasibility study, allowing a comparison of results across the two versions of the techniques.

Results/Lessons Learned: The quantitative data from this experiment allowed us to:

- Verify the difference between types of defects found by horizontal and vertical techniques.
- Show that having expertise in the domain was not helpful for subjects in the design inspection.
- Show that being a participant in a requirements inspection for the same system did not improve a subject's performance in the design inspection.

But, the qualitative data provided us with some potential ways of improving the techniques:

- Order of dealing with information must match the subjects' own way of thinking about the problem.
- Amount and type of training necessary needed to be modified.
- Differences in design approaches could affect design inspection.

In contrast to the global results that were obtained from the feasibility studies, the results from this study are more detailed. We began to understand the impact of the individual steps and their ordering on the performance of the inspectors. Also, we were able to get a better understanding about how domain expertise might or might not have an impact on the inspection.

This led us to produce a third version of the techniques, using the data from the observations about the way that readers applied the techniques. This version of the techniques also focused more on the semantics behind the design models and less on the syntax. We also changed terminology, from "defects" to "discrepancies", reflecting the fact that inspectors and designers may have different ideas about the design. Additional improvements were made regarding training and discrepancy report forms. The details of the process evolution up to this point (along with the third version of the techniques) are presented in a technical report [16]. This technical report shows excerpts from the third version of horizontal and vertical reading techniques. These techniques can be compared with the previous ones presented in [20] to observe the evolution based on these study results.

3.3 Identifying Usability in the Context of a Software Life Cycle

Previous studies had convinced us that the techniques were feasible, in that their use could detect defects and that their individual steps and ordering seemed reasonable. However, we still had no evidence that they could be used as part of a software development project, i.e. that they did not require a prohibitive amount of effort, that what they required was available in a typical development environment, and that their effects were useful for continuing the development of a system. For this understanding, we performed two case studies to evaluate the techniques inside of a software development process in a classroom environment. The first, described in Section 3.3.1, was done at UMCP during the Spring 2000 semester. Here the OORTs were used in a waterfall lifecycle method. The second one, described in Section 3.3.2, was done at the University of Southern California (USC) during the Spring 2001 semester. Here the OORTs were used in a Spiral lifecycle model in the context of a Fagan-style inspection process [6].

3.3.1 Lifecycle Case Study 1: UMCP

Subjects: The subjects came from a senior level undergraduate software-engineering course. Of the 42 students in the class, 14% had some previous experience with OO design in industry while 45% had classroom experience with OO design but had not used it on a real system (40% had no prior experience in OO design). All the students were trained in OO development, OO software design

activities, UML, and OO design inspections as part of the course. The subjects were grouped into high, medium, and low expertise categories, and one person from each group was randomly assigned to each of the 14 3-person teams.

Material: An evolved version of the techniques based on the results from the previous study was applied in the evolution of the PGCS system, described in the previous section. The students were required to add functionality to that system that allowed customers to reserve tickets and pay bills over the Internet.

Procedure: The subjects used a waterfall development process, to create an enhanced version of an existing system. Prior to the study discussed here, the subjects had created and inspected the requirements document for the complete PGCS system. After correcting the defects found during the requirements inspection, each team created a design for the system.

Once the initial design had been created, all teams used the horizontal reading to inspect their own designs to ensure that they were consistent. They corrected any defects that they found. After the designs had been corrected, the teams traded designs. Each team then performed the vertical reading techniques on a design for another team. The list of discrepancies found by the reviewers was then returned to the authors of the design for correction. In both of these inspections, team meetings followed individual review.

In the overall scope of the software development process there was no control group here. This occurred for two reasons: first, the design inspection was one small part of a larger experiment, and the overall experimental design did not allow for a control group. Secondly, in a classroom environment, it was not possible to provide instruction on a topic to only a portion of the class.

Data Collection: Questionnaires and an analysis of the defect lists were used to evaluate the effectiveness of the techniques in the development process. The questionnaires were used throughout the development cycle to collect both qualitative and quantitative data. The quantitative data collected include both background information, used to classify the subjects as having high, medium, or low expertise, and the amount of time taken to use the techniques, used to evaluate feasibility of use. The qualitative data collected by the questionnaires concerned:

- Opinions of the helpfulness of the techniques.
- Problems encountered using the techniques, or extra knowledge that was needed to use the techniques.
- Opinions of effectiveness of training.

Analysis of the defect lists provided quantitative data about the number and types of defects found by the teams. The data was useful in determining if the output of the reading process uncovered defects and was useful for continuing the development process.

Results/Lessons Learned: Here our results continued to get even more specific, since the qualitative data from this, our first case study, provided us with some lessons about the techniques and how they fit with other development processes. First, we found that not only were the subjects able to apply the techniques, but also there was no difficulty in their interaction with the lifecycle and other processes (specification, design, implementation, and testing) used in this development environment. Second, whereas in earlier studies we found that the techniques were useful for finding defects, here we verified that the defects being reported were of sufficient importance that their correction did lead to improved system quality (i.e. the issues reported did represent real and nontrivial problems with system quality). In addition they turned out to have another use: The vertical techniques helped students to gain a better understanding of the system functionality and how it should be represented in the design. Third, not only were the techniques feasible to use, but the effort required was not prohibitive compared to other system tasks; the design inspections required on average 20 hours per team, or 24% of the overall effort spent on design. Fourth, outside of the training in the techniques,

the subjects required no special knowledge that was not previously gained during the development of the system.

Finally, we found the techniques to be useful for teaching OO design. Specifically, we were able to use the horizontal techniques to improve the OO training. While we expected the subjects to use the techniques to look for defects in the designs, they found that what they were instructed to look for in terms of defects gave them a good idea of things that would not appear in a quality design.

3.3.2 Lifecycle Case Study 2: USC

Subjects: The subjects were members of a graduate level software engineering class. More than 50% of the students had industrial experience developing software, while only one student had no experience at all. Only 25% of the subjects had industrial experience with OO design, with 45% more having classroom experience. The students were trained in OO design and OO design inspections as part of the course.

Material: Based on discussions with a local expert in the Spiral lifecycle model and the MBASE documentation standard [3][4] used on the projects, the OORTs were tailored. The subjects used this tailored version of the OORTs on their projects. Each team was working on a different project with real customers, mainly in the domain of digital library applications. Most of the designs ranged from 10 to 20 classes in the class diagram. Because this was a two-semester project, these projects were larger than the projects in the case study from Section 3.3.1.

Procedure: The subjects were using the Spiral development model to create their software. The OORTs were used in one development iteration to aid in the inspection of the designs.

The subjects used Fagan-style inspections in their projects. Unlike the other studies, the goal of the individual review was to prepare the individual reviewers for the main defect detection effort, which occurred at the team meeting. So, the individual inspectors used the OORTs in the preparation phase to help them make a list of potential defects, which they wanted to discuss during the team meeting.

As in previous experiments, there was no control group. It was not possible, based on constraints of the class, to divide the class into a control and an experimental group. Also, pedagogically we could not teach the OORTs to only part of the class.

Data Collection: Questionnaires and defect lists were used to evaluate the effectiveness of the techniques. The questionnaires were used to collect both qualitative and quantitative data. The quantitative data included the background and experience of the subjects, as well as the defects they reported. The qualitative data included:

- Opinions of effectiveness and usefulness of the techniques in the spiral lifecycle model and with the MBASE guidelines.
- Opinions of effectiveness of the training

Results: The data from this study showed us that the subjects were able to find defects using the OORTs. Correction of these defects helped the subjects in their projects. The subjects also reported that they found the techniques useful and that the time required was not prohibitive in the context of the whole project. Most subjects thought the techniques were useful enough to recommend that they be used again.

3.4 Using in Industrial Environments

The series of classroom studies had provided a body of evidence that, first, yielded a proof-of-concept of the usefulness of the process and second, identified a set of issues that are important for tailoring this process for effective industrial use. For example, we have some experience with

modifying the techniques for use in different development methodologies (e.g. spiral vs. waterfall) and different inspection models (e.g. Fagan inspections).

To demonstrate the feasibility of using OORT's in industrial environment, Melo et al. [11] describe a case study that applied the techniques to guide inspections in a professional environment. The application domain is control of tax collection (for a Brazilian state government). The system's aim is to allow commercial tax declarations for merchants and services to be submitted using the Internet. **Subjects**: A team of 5 people from different development departments performed the inspection. The inspectors all had some UML knowledge, although with varying levels of expertise. They also received a tutorial about inspections.

Material: The version of the techniques described in Section 3.2 was used. The project artifacts (use cases, class description, class and sequence diagrams) were prepared by the client. The independent inspectors received these artifacts along with other material for accomplishing the inspection (inspection forms and discrepancy report forms).

Procedure : One reviewer, with a better understanding of the OORTs, was present in all reviews so that the other inspectors had a resource to answer any questions that came up about the process.

Data Collection: The inspectors collected any issues they noticed during the review on the defect report forms. After the inspection, they met to discuss the possible problems in the project artifacts and to produce a final list of defects.

Results: As reported by Melo et al [11], inspectors were able to find (on average) 35 defects. The inspector with the highest expertise with inspections and the OORTs found 57 defects. The inspector who was least familiar with the techniques was able to find 12 defects. Across all inspectors, the number of false positives was low. The inspectors suggested this was due to the participation of a very effective moderator. The average time spent during the inspection (individual review plus meeting) of the artifacts was 5.8 hours, and the average time per defect reported was 15 minutes.

At this point in time, having run a case study in a classroom environment and a case study in an industrial environment, our next step is to run another industrial case study to make sure these ideas can be tailored and transferred in additional, different industrial environments. For this we are currently seeking an industrial partner who would be interested in receiving training in the techniques and allow us to assist in, and study, their use on a project.

4. OORTs Usage

In this section, we describe the design issues relevant to horizontal and vertical reading, and provide some guidelines for the practical use of the OORTs, based on observations on their use over the series of evaluative studies.

The main idea in applying horizontal reading is to understand whether all the high-level design artifacts are representing the same system. We must keep in mind that the artifacts should model the same system information but from different perspectives. UML organizes the artifacts and different types of information based on the type of system information they contain. There are specific artifacts to capture essentially static information (basically, the structure assumed by the domain's objects while playing specific roles in the problem domain) and specific artifacts to capture essentially dynamic information (basically, the consequences when objects are asked to behave in order to accomplish system functionalities). These different views allow developers to understand the objects from complementary points of view. However, these differences among the diagrams make the inspection process a bit more complicated. For instance, when comparing sequence diagrams against state machine diagrams two different perspectives must be combined to interpret and identify possible defects. Each one of the sequence diagrams represents some system objects and the messages exchanged between them that implement some functionality required by the user while, on the other hand, the state machine diagram is a picture of what happens to one object when it is

influenced by the events occurring in multiple sequence diagrams. Sequence diagrams show the specific messages exchanged by objects, while state diagrams show how the system responds to events, which can be messages, services, or functionality. Both diagrams must convey information about conditions and constraints on the functionality. So, the horizontal reading techniques explore these types of differences and help reduce the semantic gap between the documents. See Appendix B.1 to get the complete version 3 horizontal reading techniques descriptions in English and Appendix C.1 for the equivalent set in Portuguese.

To apply vertical reading readers should be aware of the differences between the two lifecycle phases in which the documents were created and how the traceability between these two different phases could be explored. The levels of abstraction and information representation between these phases are quite different. Requirements and use cases should precisely describe the problem and thus use a totally different representation than the design artifacts. There is no direct mapping from one phase (specification) to another (design). Vertical reading techniques explore such ideas and provide some guidance to help the reader identify the information s/he needs. For example, the requirements descriptions and use cases capture the functionality of the entire system and in some cases the services, but not the messages. Designers using these requirements and use cases decide about the messages based on the viewpoint (abstraction) used to classify and organize the classes. Sequence diagrams are organized based on messages that work together in some way to provide the services, which compose the required functionality. Requirements and use cases describe constraints and conditions in general terms; on a sequence diagram such information must be made explicit and associated with the appropriate messages. So, vertical reading techniques explore these types of differences by defining some guidelines for tracing the right information between these two lifecycle phases. See Appendix B.2 to get the complete version 3 vertical reading techniques descriptions in English and Appendix C.2 for the equivalent set in Portuguese.

To support these two types of reading (horizontal and vertical) we have introduced some new terminology to describe the actions of the system. First, because the level of abstraction and granularity of the information in the requirements and use-cases is different from the abstraction and information in the design artifacts, the concept of *system functionality* was broken down into three complementary concepts (messages, services, and functionality). 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 be associated with class behaviors. Services are combinations of one or more messages and usually capture some basic activity necessary to accomplish a functionality. They can be considered low-level actions performed by the system. They are the "atomic units" out of which system functionalities are composed. A service could be used as a part of one or more functionalities. We use the term "functionality" to describe the behavior of the system from the user's point of view, in other words, the functionality that the user expects to be visible. A functionality is composed of one or more services. Users do not typically consider services an end in themselves; rather, services are the steps by which some larger goal or functionality is achieved. For example, highlighting a block of text in a document would be considered a service. That service could be used in conjunction with various other services to accomplish different functionalities. To accomplish the functionality of making a section of text bold, the 'highlight' service would be combined with the service of making selected text bold. On the other hand, the 'highlight text' service could be combined with the 'cut' service to accomplish the functionality of removing an entire section of text.

A second important piece of terminology is that of conditions and constraints. A condition describes what must be true for one or another functionality to be executed. A constraint constrains system functionality. It must always be satisfied for system functionality accomplishment. This information is important to readers comparing different diagrams since it describes *how* the functionality must be implemented; this information is important to maintain with the functionality it describes.

To organize the reading process, reading responsibilities can de distributed among the members of the inspection team, reducing the reading effort per team member and improving the reading process. In this way, each one of the readers can apply a reduced number of reading techniques, or even deal with a reduced number of artifacts at the same time. After individual review, it is important to organize a meeting in order to review each one of the individual defect lists and to create a final list that reflected a group consensus of the defects in the documents. It is not necessary to apply the techniques in a particular order, but it seems to be reasonable to apply first horizontal reading for all existing design artifacts and then vertical reading, to ensure that a consistent system description is checked against the requirements. In Figure 3 is an example of how the techniques could be organized among a team of three reviewers.

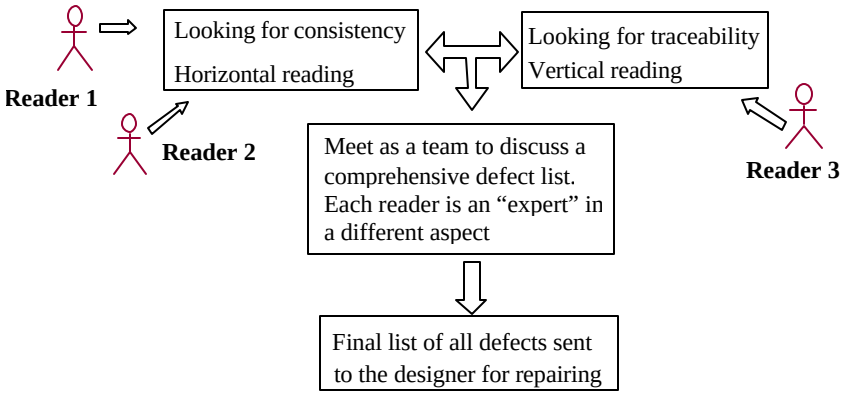


Figure 3 – Organizing reading with 3 readers

5. ONGOING WORK

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 version of the reading techniques keeps its feasibility and interest. We have found observational techniques useful, because they have allowed us to follow the reading process as it occurred, rather than trying to interpret the readers' post-hoc answers as we have done in the past. Observing how readers normally try to read diagrams challenged many of our assumptions about how our techniques were actually being applied.

However, one question is still open in this area. It regards the level of automated support that should be provided for such techniques. The observational studies have allowed us to understand which steps of the techniques can feel especially repetitive and mechanical to the reader. So, the clerical activities regarding the reading process using OORTs must be precisely defined and identified. For this situation, further observational studies play an important role and they should be executed aiming to collect suggestions on how to automate the clerical activities concerned with OORTs.

So far, the techniques have been used in different contexts and by more than 150 different developers, at different levels of expertise from academia and industry. Additionally, replications by independent researchers have begun to take place in different companies and research groups. We have made arrangements for our conclusions, future technical publications, and some data from different environments to be available through the national Center for Empirically- Based Software Engineering (CeBASE), at www.cebase.org.

The results we have so far have shown that the techniques are ready to be used in real projects. Still, we are interested in the application of the techniques in various development environments, using reviewers with different levels of experience and with different development paradigms (e.g.

waterfall, spiral, etc.). We are not advocating the techniques as a “one size fits all” process, but understand that tailoring needs to be done for various different environments. We are continuing to work in this area to enhance the practicality and feasibility of the techniques for industry. The feedback from users and the observation of their effectiveness are playing an important role as we work towards a useful and feasible set of reading techniques for OO design.

6. ACKNOWLEDGEMENTS

This work was partially supported by UMIACS and by NSF grant CCR9706151.

Our special thanks to Dr. Walcélio Melo, who applied the techniques in the context of an industrial project. We recognize the support, management and dedication of Prof. Victor R. Basili for this research work. Dr. Travassos also recognizes the partial support from CAPES - Brazil while joined to the Experimental Software Engineering Group at the University of Maryland/College Park.

7. REFERENCES

[1] Basili, V. R.; Green, S.; Laitenberger, O.; Lanubile, F.; Shull, F.; Sorumgard, S. and Zelkowitz, M. V. (1996) The Empirical Investigation of Perspective-Based Reading, Empirical Software Engineering Journal, I, 133-164.

[2] Basili, V.; Caldiera, G.; Lanubile, F. and Shull, F. (1996b). Studies on reading techniques. *In Proc. of the Twenty-First Annual Software Engineering Workshop*, SEL-96-002, pages 59-65, Greenbelt, MD, December.

[3] Boehm, B. A Spiral Model of Software Development and Enhancement, *IEEE Computer*, May 1988, pp. 61-72.

[4] Boehm, B., Port, D., Abi-Antoun, M., and Egyed, A. Guidelines for the Life Cycle Objectives (LCO) and the Life Cycle Architecture (LCA) deliverables for Model-Based Architecting and Software Engineering (MBASE). USC Technical Report USC-CSE-98-519, University of Southern California, Los Angeles, CA, 90089, February 1999.

[5] Fagan, M. E. (1976). "Design and Code Inspections to Reduce Errors in Program Development." IBM Systems Journal, 15(3):182-211.

[6] Fagan, M. (1986). "Advances in Software Inspections." IEEE Transactions on Software Engineering, 12(7): 744-751, July.

[7] Fowller, M.; Scott, K. (2000). UML Distilled: Applying the Standard Object Modeling Language, Second edition, Addison- Wesley. ISBN 0-201-65783-X

[8] Fusaro, P.; Lanubile, F. and Visaggio, G. (1997). A replicated experiment to assess requirements inspections techniques, *Empirical Software Engineering Journal*, vol.2, no.1, pp.39-57.

[9] Gilb , T. and Graham, D. (1993). Software Inspection. Addison-Wesley, reading, MA.

[10] Jacobson, I.; Christerson, M.; Jonsson, P. and Overgaard, G. (1995). Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley, revised printing.

[11] Melo, W.; Shull, F. and Travassos, G.H. (2001). Software Review Guidelines. Technical Report ES-556/01. Systems Engineering and Computer Science Program. COPPE. Federal University of Rio de Janeiro. September.

[12] NASA. (1993). National Aeronautics and Space Administration, Office of Safety and Mission Assurance. "Software Formal Inspections Guidebook". Report NASA-GB-A302, August 1993.

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

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

[15] Shull, F.; Travassos, G. and Basili, V. (1999). Towards Techniques for Improved OO Design Inspections. Workshop on Quantitative Approaches in Object-Oriented Software Engineering (in association with the 13th European Conf. on Object-Oriented Programming), Lisbon, Portugal. On line at http://www.cs.umd.edu/projects/SoftEng/ESEG/papers/postscript/ecoop99.ps.

[16] Shull, F.; Travassos, G. H.; Carver, J. and Basili, V. R. (1999b). Evolving a Set of Techniques for OO Inspections. Technical Report CS-TR-4070, UMIACS-TR-99-63, University of Maryland, October. http://www.cs.umd.edu/Dienst/UI/2.0/Describe/ncstrl.umcp/CS-TR-4070

[17] Shull, F.; Rus, I. and Basili, V. (2000). How Perspective Based Reading can Improve Requirements Reading. IEEE Computer, July.

[18] Shull, F.; Carver, J.; and Travassos, G.H. (2001). An Emperical Methodology for Introducing Software Processes. In *Proceedings of European Software Engineering Conference*, Vienna, Austria, Sept. 10-14,2001. p. 288-296.

[19] Travassos, G.; Shull, F.; Fredericks, M., and Basili, V. (1999). Detecting Defects in Object-Oriented Designs: Using Reading Techniques to Improve Software Quality. In the Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Denver, Colorado.

[20] Travassos, G. H.; Shull, F. and Carver, J. (1999b). Evolving a Process for Inspecting OO Designs. XIII Brazilian Symposium on Software Engineering: *Workshop on Software Quality.* Florianópolis, Curitiba, Brazil, October.

[21] Travassos, G. H.; Shull, F.; Carver, J. and Basili, V. R. (1999c). Reading Techniques for OO Design Inspections, 24th Annual Software Engineering Workshop, NASA/SEL, Greenbelt, USA, December. On line at http://sel.gsfc.nasa.gov/website/sew/1999/topics/travassos_SEW99paper.pdf.

[22] Travassos, G. H.; Shull, F. and Carver, J. (2000). A Family of Reading Techniques for OO Design Inspection. XIV Brazilian Symposium on Software Engineering: *Workshop on Software Quality*. João Pessoa, Paraíba, Brazil, October.

[23] Travassos, G.H.; Shull, F. and Carver, J. (2001). Working with UML: A Software Design Process Based on Inspections for the Unified Modeling Language, Advances in Computers, 54(35-97), Academic Press.

[24] Votta Jr., L. G. (1993). "Does Every Inspection Need a Meeting?" ACM SIGSOFT Software Engineering Notes, 18(5): 107-114, December.

[25] Zhang, Z.; Basili, V. and Shneiderman, B. (1998). An empirical study of perspective-based usability inspection. Human Factors and Ergonomics Society Annual Meeting, Chicago, October.

APPENDIX A – DEFINITIONS AND DIAGRAM EXAMPLES

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.

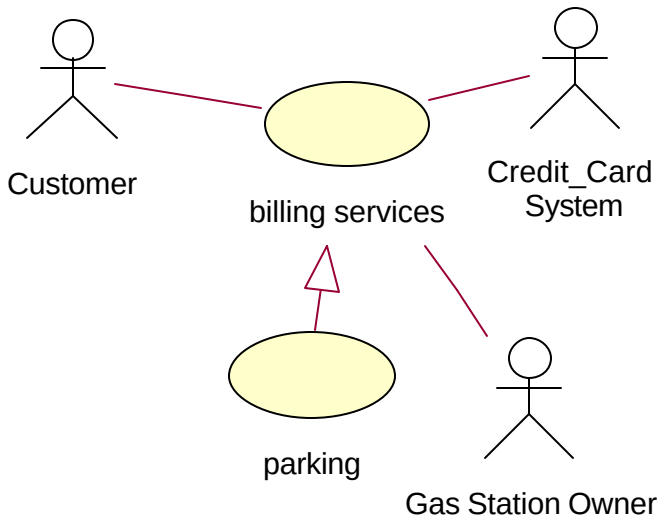
For example, consider the example diagrams provided in the appendix D. In example 2, the sequence diagram describes how classes collaborate to provide some *functionality*: the ability to lease a parking spot. This functionality is meant to describe a use of the system from the user's point of view; 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.

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.

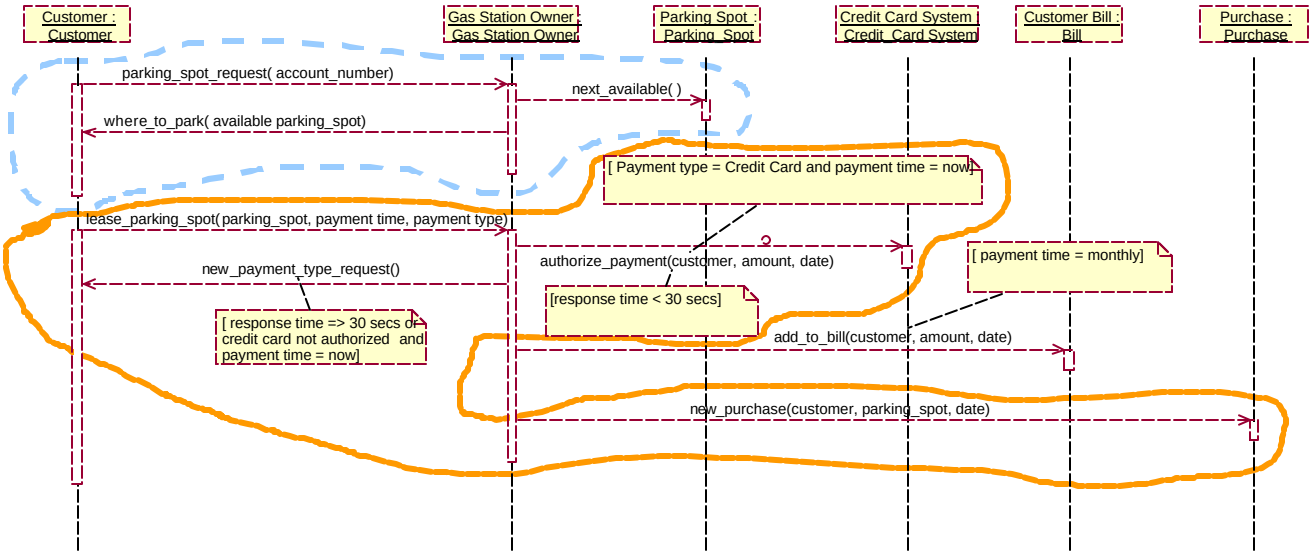
Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland



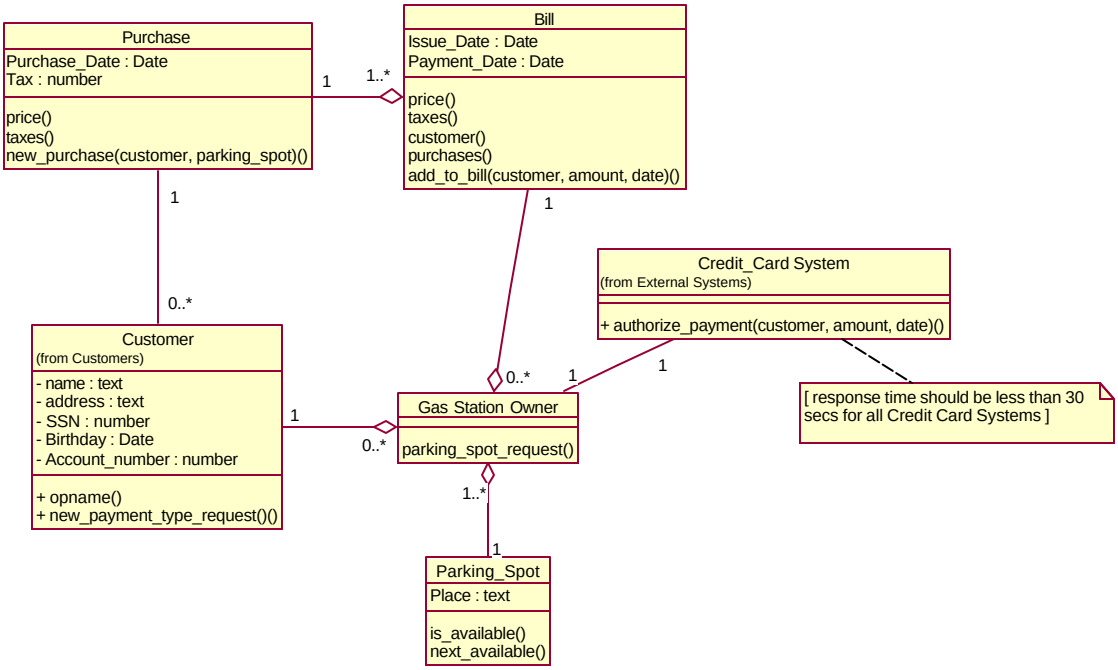
A customer, giving his account_number, asks the Gas Station Owner for an available parking spot to park his car.
To get an available parking spot Gas Station Owner searches for the next parking place available. With this information the customer can confirm the lease of the parking place. The time of 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.

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

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.



Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

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, car maintenance: US$ 150.00 and parking spot: US$ 5.00 per day). The tax is 5% added to the final price of the purchase. Sometimes, the Gas Station owner can define discounts to those prices.

Class Description

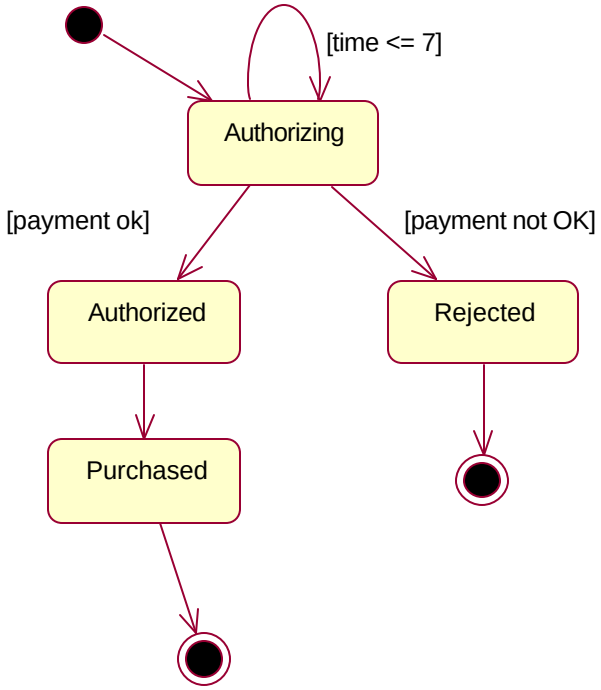
Class name: Purchase
Category: Customers
External Documents:
Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: none
Public Interface:
Operations:
price
taxes
Private Interface:
Attributes:
Purchase_Date : Date
Tax : number
Service: Services
Implementation:
Attributes:
Purchase_Date : Date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

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:
Price : number = 150.00

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

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.

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

APPENDIX B.1 – OORT's 3.0 – Horizontal Reading – English Version

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.

I. **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).

A. For each sequence diagram, underline the system objects and classes, and any actors, with a blue pen.

B. 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.

C. 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.

II. **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.

A. 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. Check for the following discrepancies and mark on the **discrepancy report form** :

1) **If a class or object cannot be found on the class diagram, it means that the information is inconsistent between both documents, it is present in one and absent in the other.**
2) **If an actor cannot be found, determine whether that actor needs to be represented as a class to perform the necessary behavior. If it does, then information that is present in the sequence diagram is missing from the class diagram.**

B. 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

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

to find the necessary features. Finally, verify that for each service, the messages described by the sequence diagram are sufficient to achieve that service. Check for the following discrepancies, and mark on the **discrepancy report form**:

1) **Make sure that for each message on the sequence diagram the receiving class contains an appropriate behavior on the class diagram. If not, it means that there is an inconsistency between the diagrams. A behavior is present in the sequence diagram, but missing on the class diagram.**
2) **Make sure that there are appropriate behaviors for the system services? If not, there is a service present on the sequence diagram that is not represented on the class diagram.**
3) **Make sure there is an association on the class diagram between the two classes between which the message is sent. If not, an association is present in the sequence diagram, because of the message exchange, but not present in the class diagram.**
4) **Make sure that there are not any behaviors missing, which would prevent the service from being achieved. If there are, it means that something is missing from the sequence diagram.**

C. Verify that the constraints identified in the sequence diagram can be fulfilled according to the class diagram. Check for the following discrepancies, if any of the following statements are not true then information on the sequence diagram has not been represented in the class diagram. Mark this on the **discrepancy report form**.

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 this information is included on the class diagram. (It may be as a constraint on a class or relation on the class diagram or 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 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.)**

D. 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?). Check for the following discrepancies:

1) **Make sure that it is logical for the class to receive this message with these data.**
2) **Make sure you can verify that the constraints are feasible.**
3) **Make sure all of the necessary attributes are defined. If not, the diagrams may contain incorrect facts.**
4) **For the classes specified in the sequence diagram, make sure the behaviors and attributes specified for them on the class diagram make sense.**
5) **Make sure the *name* of the class is appropriate for the domain, and for its attributes and behaviors.**
6) **Make sure the relationships with other classes are appropriate.**
7) **Make sure the relationships are of the right *type.* a(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.**

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

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. State diagrams 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:

I. 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.

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

1) **If you can't determine the class that is being modeled, then something has been omitted or is ambiguous. Indicate this on a discrepancy report form.**

B. 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.

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

D. 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.

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

1) **Make sure that you can understand and describe what is going on with the object just by reading the state machine. If you cannot, then the state machine is ambiguous. Indicate this on the discrepancy report form.**

II. 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.

A. Use the class description to find the class or class hierarchy that corresponds to this state diagram.

1) **If you can't find the corresponding class fill out a discrepancy report form because you have found an inconsistency. The state machine describes a class that has not been described on the class description.**

B. 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".)

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

- 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.

    1) **If there are any unstarred states then something is missing from the class description. If you can determine from your semantic knowledge of the domain, that the extra state does not make sense, then indicate this on the discrepancy report form, otherwise just indicate that the two diagrams are inconsistent.**

C. 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.

Check for the following discrepancies, and fill out a **discrepancy report form** if you find any:

1) **Make sure that all actions are encapsulated by the class description. If they are not, then something is represented in the state diagram, but not in the class description.**
2) **Make sure that all of the constraints are encapsulated by the class description. If they are not, then something is represented on the state diagram, but not in the class description.**
3) **Make sure all of the data need to verify a constraint is present in the class description. If it is not all there, then you have found information in the state diagram that is not in the class description.**

III. Compare the class description 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.

A. 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.

1) **Using your semantic knowledge of this class and the behaviors it should encapsulate, make sure that all states are described. If not, something is missing and the class as described cannot behave, as it should. Indicate this on a discrepancy report form.**

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

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:

I. **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.

A. Determine which class is being modeled by this state diagram.
   1) **If you can't determine the class that is being modeled, then something has been omitted or is ambiguous. Indicate this on a discrepancy report form.**

B. 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.

C. 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, …].

D. Think about the states and actions you have just identified, and how they fit together.
   1) **Make sure that you can understand and describe what is going on with the object just by reading the state machine. If you cannot, then the state machine is ambiguous. Indicate this on the discrepancy report form.**

II. **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.

A. 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.
   1) **If there are no sequence diagrams that have this class in them, then fill out a discrepancy report because there is information in a state diagram that does not appear on the sequence diagrams.**

For each sequence diagram identified in the previous step:

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

C. 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.

D. 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.
   1) **Make sure, semantically, that you could do this mapping. If you cannot, then there are messages needed for a state transition that are not in the sequence diagram. Fill out a**

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

discrepancy report form, because information included in one diagram is not included in the other one.

E. 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). If any of the following occur, then fill out a **discrepancy report form**:

1) **Make sure that you can find a correspondence between the conditions and constraints on the state and sequence diagrams. If not, then one diagram has information that is not on the other.**
2) **For the information that appears on both diagrams, make sure that it is consistent. If it is not, then you have found the same information represented on two different diagrams in an inconsistent way.**

III. 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.

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

1) **If the transition action was labeled with a constraint, see if you can find a message or sequence of messages capable of satisfying the constraint. If not, you have found information represented in one diagram but not in the other. The state diagram requires system services that are not described on any sequence diagram. Fill out a discrepancy report.**
2) **If the transition action was labeled with an event, see if you can find a message, a sequence of messages, or some event performed by an actor that achieves the transition action. If not, you have found information represented in one diagram but not in the other. The state diagram requires system services that are not described on any sequence diagram. Fill out a discrepancy report.**

B. 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.

1) **If the order does not match, the fill out a discrepancy report form, information is represented on two diagrams, but in an inconsistent way.**

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

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:

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

I. 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:

A. Find the relevant class description. Mark the class on the class description with a blue symbol (*) when found.

1) **If you can't find the description, fill out a discrepancy report form, because a class present on the class diagram is not present in the class description.**

B. 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.

1) **Using your knowledge, make sure you can 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 reporting a discrepancy because: *outside knowledge*.**

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

1) **Make sure that the same set of attributes is present in both the class description and the class diagram. If not, fill out a discrepancy report form because information is present in one document but not present in the other.**

2) **Make sure this class can *meaningfully* encapsulate all these attributes, that is, does it make sense to have these attributes in the class description, and that the basic types assigned to the attributes *feasible* according to the description of the attribute. If not, fill out a discrepancy report form indicating a discrepancy because: *outside knowledge*.**

D. Verify that all the behaviors and constraints are described.

1) **Make sure the same set of behaviors and constraints is present in both the class description and the class diagram, and that they use the same style or level of granularity (e.g. pseudocode) to describe the behaviors. If not, then information on one diagram is not present on the other, or it is inconsistent between the two.**

2) **Make sure this class can *meaningfully* encapsulate all these behaviors. Make sure the constraints make sense for this class. Make sure that behaviors can accomplish their tasks using the attributes that have been defined (for this or some other class). If not, fill in a discrepancy report indicating a discrepancy because: *outside knowledge*.**

3) **Make sure the constraints are satisfiable using the attributes and behaviors that have been defined. If not, you have found a situation where the behaviors and constraints as defined cannot be satisfied using the attributes and behaviors that have been defined. Indicate this on a discrepancy report form as a discrepancy because: *outside knowledge*. Describe the situation.**

4) **Make sure that the behaviors for this class do not 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**

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

functionality addressed to determine if such reliance is really necessary.) If they do, then you have found a possibly poor design situation. Fill out a discrepancy report form indicating this situation.

E. If the class diagram specifies any inheritance mechanisms for this class**,** verify that they are correctly described.

1) **Make sure the inheritance relationship is included on the class description. If it is not, fill out a discrepancy report form. Information on the class diagram is not on the class description.**

2) **Use the class hierarchy to find the parents of this class. Make sure that, *semantically*, a <class name> is a type of <parent name>, and that it makes 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: *outside information*.**

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

1) **Make sure that the object roles are captured on the class description, and that the correct graphical notation is used on the class diagram. If you find a problem, fill out a discrepancy report form indicating if information is omitted in one diagram, or if the notation is incorrect.**

2) ***Semantically*, make sure 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 they don't make sense then you have uncovered a potential style issue: the relationships should not be defined in this way. Fill in a discrepancy report describing the problem: *outside information*.**

3) **If cardinalities are important, make sure they are described in the class description. Given your understanding of the relationship, make sure the quantities of objects used are enough. If not, fill in a discrepancy report because information in one diagram is not present in the other.**

4) **Make sure that there is some attribute representing the relationship. If not, fill in a discrepancy report indicating that information in one diagram is not present in the other.**

5) **Make sure that the relationship uses a feasible basic type or structure of basic types (if multiple cardinality is involved). If not, fill in a discrepancy report form indicating a discrepancy because: *outside information*.**

II. Review the class descriptions for extraneous information.

INPUTS: Class description.

OUTPUTS: Discrepancy reports.

A. Review the class descriptions to make sure that all classes described actually appear in the class diagram.

1) **Make sure there are no unstarred classes on the class description. If there are any, fill out a discrepancy report form because a class on the class description is not present on the class diagram.**

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

Discrepancy Report Form for Horizontal Reading

Name of the project:_____ Team: _____ Horizontal reading technique: _____

Inspection starts: _____ (time) Date: _____(date)

Documents that are been read [fill in name and type]:
Document 1: _____ Document 2: _____

Type of Concept:
(AC) actor (AT) attribute (BE) behavior (CA) cardinality
(CO) condition (CR) Constraint (DA) data (IN) inheritance
(ME) message (OB) object/Class (RE) relationship (RO) role

Discrepancy type (Disc. Type):
(1) present in Document 1 but not Document 2
(2) present in Document 2 but not Document 1
(3) present in both documents but inconsistent or ambiguous
(4) present in both documents but using an incorrect representation or notation
(5) present in both documents but extraneous
(6) missing in both documents [explain below]

Severity (Sev.):
(NS) Not serious. But needs to check this document.
(IN) This discrepancy invalidates this part of the document. Check both documents.
(SE) Serious. It's not possible to continue the reading of this document. It should be redesigned.

Fill in the table with the discrepancies found:

Disc.#	Type of concept	Name	Disc. type	Sev.	Comments
01					
02					
03					
04					
05					
06					
07					
08					
09					
10					
11					
12					
13					
14					
15					

(Use backside if necessary)

Inspection end: _____ (time)

Use the following template to detail some found discrepancy (all the serious, 5 and 6 type discrepancies must be explained) that you consider be necessary to be explained:
Discrepancy number (the same number used in the table): xx
Description:

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

Disc.#	Type of concept	Name	Disc. type	Sev.	Comments
16					
17					
18					
19					
20					
21					
22					
23					
24					
25					
26					
27					
28					
29					
30					
31					
32					
33					
34					
35					
36					
37					
38					
39					
40					
41					
42					
43					
44					
45					
46					
47					
48					
49					
50					
51					
52					
53					
54					
55					

(Use additional tables if necessary)

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

APPENDIX B.2 – OORTs 3.0 – Vertical Reading - English Version

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 is necessary in the final system.
2. A set of class descriptions that lists the classes of a system along with their attributes and behaviors.

I. 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).

A. Read over the each functional requirement to understand the functionality that it describes.

B. 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.

C. 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.

D. 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.

II. 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.

A. 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 (*).

1) **Make sure the classes receive the right information for accomplishing the required behaviors. Make sure *feasible* results are produced. If not, the classes cannot implement the functionality appropriately. Indicate this on the discrepancy report form and mark whether it is because of omitted functionality or incorrect or ambiguous information.**

B. 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.

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

C. 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 (*).

1) **Make sure the class descriptions contain sufficient information regarding the concepts that play some role in this functionality and the class names have some connection to the nouns you had marked. If not, or if the classes are using ambiguous information to describe the concepts indicate this on the discrepancy report form.**

2) **Make sure these classes encapsulate (blue-marked) attributes concerned with the nouns you marked and (green-marked) behaviors concerned with the verbs or actions descriptions you had marked. Also make sure that all identified constraints and conditions for these classes regarding this requirement are described. If not, you have found important information from the requirements omitted from the design. Indicate this on the discrepancy report form.**

D. 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 (*).

1) **Make sure the class description is using *feasible* types to represent information; given the requirements description and that the (yellow-underlined) constraints and conditions on the attributes were observed in their definition. If not, you have found incorrect information in the design. Indicate this on the discrepancy report form.**

III. 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.

A. Look for descriptions of functionality in the requirements that have been omitted from the design.

1) **Make sure that there are no unstarred nouns or verbs in the requirements. If there is one, make sure that it should have been included in the design, and was not there merely for clarification. If it should have been in the design, then information has been omitted from the design. Indicate this on the discrepancy report form.**

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

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.

I. 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).

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

B. 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.)

C. 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").

D. 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.

II. 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).

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

B. 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.

C. 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.

III. 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.

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

A. 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 and the sequence diagram with a blue star (*) if it can be found on the sequence diagram.
    1) **If there are any unstarred nouns on the use case, it means that a concept was used to describe functionality on the use case but it 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.(Is this correct? We are referring to the Class Description here, but that is not part of this technique) A concept was described on the use case but has not appeared in the system design. Fill in a discrepancy report because necessary functionality has been omitted.**
    2) **If there are any unstarred nouns on the sequence diagram 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 it does not, fill in a discrepancy report because this information is extraneous.**

B. 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?
    1) **Make sure that 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 changing the order will change the functionality, fill in a discrepancy report because the information on the design is incorrect.**
    2) **Make sure that the data exchanged are all in the correct message and that the messages go between the correct classes (i.e. do the labels "Di,j" for the data match between diagrams). Make sure the messages make sense for the objects sending and receiving them, and 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.**

C. 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?
    1) **Make sure that the constraints are observed. Make sure all of the behavior and data on the sequence diagram are directly concerned with the use-case. If not, it means that the sequence diagram is using information incorrectly. Fill in a discrepancy report describing the problem.**

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

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 is 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:

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

II. **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: State Diagrams (SD)

Requirements Description (RD)

OUTPUTS: Object States (marked in blue on SD)

Adjacency Matrix

A. 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.

B. 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.

C. 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".

D. 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.

E. 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.

F. 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.

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

INPUT: Use Cases

OUTPUT: Completed Adjacency Matrix

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

B. 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.

C. 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.

IV. **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.**

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

INPUT: Requirements Description;

State Diagram (SD);

Adjacency Matrix (AM).

OUTPUT: Discrepancy Reports

A. 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. If you find any problems, fill out a discrepancy report.
    1) **Make sure you can 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 information has been omitted from the design. If so, then make sure this combination makes sense. If it does not, then the design has incorrect information in it.**
    2) **Make sure that there are no 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 information in the design is extraneous. If so, make sure that this split makes sense. If it does not then the design has incorrect information.**

B. 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.
    1) **Make sure all of the events on the AM appear on the SD. If not, information has been omitted from the design. If there are extra events on the state diagram, then the design has extraneous information in it.**

C. For each constraint that was marked on the AM, find it on the SD.
    1) **Make sure you can find all of the constraints that are on the AM. If you cannot, then information has been omitted from the design. If there are extra constraints on the state diagram, then the design has extraneous information.**

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

Discrepancy Report Form for Vertical Reading

Name of the project:_____ Team: ____ Vertical reading technique: _____

Inspections starts: _____ (time) Date: (date)

Documents that are been read [fill in name and type. Fill in Document 2 only if using reading 7]:

Document 1: _____ Document 2: _____

Type of Concept:

(AC) actor	(AT) attribute	(BE) behavior	(CA) cardinality
(CO) condition	(CR) Constraint	(DA) data	(IN) inheritance
(ME) message	(OB) object/Class	(RE) relationship	(RO) role

Discrepancy type (Disc. Type):
(1) necessary functionality or concept was omitted.
(2) the design is incorrect with respect to the requirements.
(3) how the design implements these requirements is ambiguous or under-specified.
(4) the design information is extraneous, i.e. not called for by the requirements.
(5) other design problem [explain below]

Severity (Sev.):
(NS) Not serious. But needs to check this document.
(IN) This discrepancy invalidates this part of the document. Check both documents.
(SE) Serious. It's not possible to continue the reading of this document. It should be redesigned.

Fill in the table with the discrepancies found. Describe the functionality from the requirements, using requirement numbers and page numbers if possible:

Disc. #	Type of concept	Name	Disc. type	Requirement Identification	Sev.	Comments
01						
02						
03						
04						
05						
06						
07						
08						
09						
10						
11						
12						
13						
14						
15						

(Use backside if necessary)

Inspection end: _____(time)
Use the following template to detail some found discrepancy (all the serious and 5 discrepancies might be explained):
Discrepancy number (the same number used in the table): xx
Description:

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

Disc. #	Type of concept	Name	Disc. type	Requirement Identification	Sev.	Comments
16						
17						
18						
19						
20						
21						
22						
23						
24						
25						
26						
27						
28						
29						
30						
31						
32						
33						
34						
35						
36						
37						
38						
39						
40						
41						
42						
43						
44						
45						
46						
47						
48						
49						
50						
51						
52						
53						
54						
55						
56						
57						

(Use additional tables if necessary)

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

APPENDIX C.1 – OORTs 3.0 – Horizontal Reading – Portuguese Version

Leitura 1 – Diagramas de Seqüência x Classes

Objetivo: Verificar se um diagrama de classes para um sistema descreve as classes e seus relacionamentos de forma que os comportamentos especificados nos diagramas de seqüência estão capturados corretamente. Para fazer isto, você verificará primeiro que as classes e objetos especificados no diagrama de seqüência aparecem no diagrama de classes. Então, você verificará que o diagrama de classe descreve os relacionamentos, comportamentos e condições que capturam a dinâmica dos serviços como estão descritos no diagrama de seqüência.

Entradas para o processo:

3. Um diagrama de classes (possivelmente dividido em pacotes) que descreve as classes de um sistema e como elas estão associadas.
4. Diagramas de seqüência que descrevem as classes, objetos, e possivelmente atores de um sistema e como eles colaboram para capturar os serviços do sistema.

III. **Pegue um diagrama de seqüência e leia-o para entender que serviços do sistema estão descritos e como o sistema deveria implementar estes serviços.**

ENTRADAS: Diagrama de Seqüência (DS).
SAÍDAS: Objetos do Sistema (marcados em azul no DS);
Serviços do Sistema (marcados em verde no DS);
Condições nos serviços (marcadas em amarelo no DS).

A. Para cada diagrama de seqüência, sublinhe os objetos do sistema e classes, e quaisquer atores, com uma caneta azul.

B. Sublinhe a informação trocada entre objetos (as setas horizontais) com uma caneta verde. Considere se esta informação representa *mensagens* ou *serviços* do sistema. Se a informação trocada está muito detalhada, para um nível de mensagens, você deverá abstrair várias mensagens juntas para entender que serviços elas estão fornecendo em conjunto. O exemplo 2 fornece uma ilustração de mensagens sendo abstraídas como serviços. Anote no diagrama de seqüência descrevendo estes serviços, e sublinhe-os também em verde.

C. Circule qualquer das seguintes restrições nas mensagens e serviços com uma caneta amarela: restrições no número de classes/objetos que uma mensagem poderia enviar, restrições nos valores globais de um atributo, dependências entre dados, ou restrições de tempo que podem afetar o estado de um objeto. Circule também quaisquer condições que determinam sob que circunstâncias uma mensagem pode ser enviada. O diagrama de seqüência no exemplo 2 mostra várias restrições e condições em mensagens. As condições relativas ao tipo de pagamento e tempo do pagamento determinam quando as mensagens *authorize_payment* e *new_payment_type_request* serão enviadas, enquanto as restrições de *response_time* para a mensagem *authorize_payment* representam restrições temporais.

IV. **Identifique e inspecione o diagrama de classes relacionado, para identificar se os objetos correspondentes do sistema estão precisamente descritos.**

ENTRADAS: Diagramas de Seqüência, com objetos, serviços e restrições marcadas;
Diagrama de Classes.
SAÍDAS: Relatório de Discrepâncias.

A. Verifique que todo objeto, classe e ator usado no diagrama de seqüência está representado por uma classe concreta no diagrama de classes. Para classes e atores, simplesmente encontre o nome no diagrama de classes. Para objetos, encontre o nome da classe da qual o objeto foi instanciado. Verifique as seguintes discrepâncias e marque-as no **Formulário de Relato de Discrepâncias**:

1) **Se uma classe ou objeto não pode ser encontrada no diagrama de classes, isto significa que a informação está inconsistente entre os dois diagramas, está presente em um e ausente em outro.**
2) **Se um ator não pode ser encontrado, determine se o ator precisa ser representado como uma classe para executar algum comportamento necessário. Se sim, então informação que está presente nos diagramas de seqüência está faltando no diagrama de classes.**

B. Verifique se para todo serviço ou mensagem marcado em verde no diagrama de seqüência, existe um comportamento correspondente no diagrama de classes. Verifique se existem comportamentos de classes no diagrama de classes que encapsulam os serviços de mais alto nível fornecidos pelo diagrama de seqüência. Para fazer isto, esteja certo que a classe ou objeto que *recebe* a mensagem no diagrama de seqüência, ou que deveria ser responsável pelo serviço, possui um comportamento associado no diagrama de classes. Esteja certo também que existe algum tipo de associação (no diagrama de classes) entre as duas classes que

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

a mensagem conecta (no diagrama de seqüência). Lembre que em ambos os casos, você pode precisar procurar na árvore de herança a qual a classe pertence para encontrar as características necessárias. Finalmente, verifique que para cada serviço, as mensagens descritas pelo diagrama de seqüência são suficientes para executar aquele serviço. Verifique as seguintes discrepâncias, e marque-as no **Formulário de Relato de Discrepância**:

1) **Esteja certo que para cada mensagem no diagrama de seqüência a classe recebedora contêm um comportamento apropriado no diagrama de classes. Caso contrário isto significa que existe uma inconsistência entre os diagramas. Um comportamento está presente no diagrama de seqüência, mas faltando no diagrama de classes.**
2) **Esteja certo que existem comportamentos apropriados para os serviços do sistema. Se não, existe um serviço presente no diagrama de seqüência que não está representado no diagrama de classes.**
3) **Esteja certo que existe uma associação no diagrama de classes entre duas classes as quais trocam mensagens. Caso contrário, uma associação está presente no diagrama de seqüência, por causa da troca de mensagem, mas não está presente no diagrama de classes.**
4) **Esteja certo que não estão faltando comportamentos, os quais poderiam evitar que algum serviço seja executado. Se existem, isto significa que algo está faltando no diagrama de seqüência**

C. Verifique que as restrições identificadas no diagrama de seqüência podem ser atendidas de acordo no diagrama de classes. Verifique as seguintes discrepâncias, se quaisquer das declarações seguintes não são verdadeiras então informação no diagrama de seqüência não foi representada no diagrama de classes. Marque-as no **Formulário de Relato de Discrepância**:

1) **Se o diagrama de seqüência descreve restrições no número de objetos que podem receber uma mensagem, esteja certo que a restrição aparece como uma informação de cardinalidade na associação apropriada do diagrama de classe.**
2) **Se o diagrama de seqüência especifica uma faixa de valores permitidos para os dados, esteja certo que uma restrição aparece como uma faixa de valores no atributo do diagrama de classes.**
3) **Se o diagrama de seqüência contêm informação relacionada às dependências entre os dados ou objetos (e.g. "um objeto 'conta' não pode existir a menos que um objeto 'compra' exista") esteja certo que está informação está incluída no diagrama de classes. (pode ser como uma restrição na classe ou relação no diagrama de classes ou pelas restrições de cardinalidade nos relacionamentos)**
4) **Se o diagrama de seqüências contém restrições de tempo que poderiam afetar o estado de um objeto (e.g. "se nenhuma entrada é recebida dentro de 5 minutos então a janela deveria ser fechada") esteja certo que esta informação está incluída como uma restrição numa classe ou relação do diagrama de classes (Por exemplo, o diagrama de classes do Exemplo 3 contêm uma restrição de tempo para a classe *"Credit_Card_System"* desde que aplica para todas as instâncias desta classe. As expressões condicionais do Exemplo 2 não deveriam aparecer no diagrama de classes porque elas não afetam o estado da classe.)**

D. Finalmente, para cada classe, mensagem e dado identificado acima, pense se, *baseado em sua experiência prévia*, isto resulta num projeto viável. Por exemplo, pense sobre os atributos de qualidade do projeto tais como coesão (todos os comportamentos e atributos de uma classe realmente pertencem a ela?) e acoplamento (os relacionamentos entre as classes são apropriados?). Verifique as seguintes discrepâncias:

1) **Esteja certo que é lógico para a classe receber esta mensagem com estes dados.**
2) **Esteja certo que você pode verificar que as restrições são viáveis.**
3) **Esteja certo que todos os atributos necessários estão definidos. Se não, os diagramas podem conter fatos incorretos.**
4) **Para as classes especificadas no diagrama de seqüência, esteja certo que comportamentos e atributos especificados para ela no diagrama de classes fazem sentido.**
5) **Esteja certo que o nome da classe é apropriado para o domínio, e para seus atributos e comportamentos.**
6) **Esteja certo que os relacionamentos com outras classes são apropriados.**
7) **Esteja certo que os relacionamentos são do *tipo* correto. (Por exemplo, um relacionamento de composição vem sendo utilizado quando uma associação faz sentido?) Se não, você encontrou um fato incorreto porque alguma coisa no projeto contradiz seu conhecimento do domínio.**

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

Leitura 2– Diagramas de Estado x Descrição de Classes

Objetivo: Verificar se as classes estão descritas de forma a capturar a funcionalidade especificada pelo diagrama de estados.

Entradas para o Processo:

3. Um conjunto de descrições de classes que lista as classes de um sistema juntamente com seus atributos e comportamentos.
4. Diagramas de estados que descrevem os estados internos que os objetos podem assumir e as possíveis transições entre estes estados.

Para cada **diagrama de estado**, execute os seguintes passos:

IV. **Leia o diagrama de estados para entender os possíveis estados de um objeto e as ações que disparam as transições entre eles.**

ENTRADAS: Diagrama de Estados (SD).

SAÍDAS: Estados do Objeto (marcados em azul no SD);
Ações de Transição (marcadas em verde no SD);
Relatórios de Discrepância.

A. Identifique que classe esta sendo modelada por este diagrama de estado.

1) **Se você não pode identificar a classe que esta sendo modelada, então alguma coisa foi omitida ou esta ambígua. Indique esta situação no Formulário de Relato de Discrepância**

B. Acompanhe a seqüência de estados e as ações de transição (trocas do sistema durante o tempo de vida do objeto que provocam uma transição de um estado para outro) através do diagrama de estado. Comece pelo estado inicial (círculo cheio) e siga as transições ate que você encontre um estado final (círculo dobrado). Esteja certo que você passou por todas as transições.

C. Sublinhe o nome de cada estado, a medida que você o alcance, com uma caneta azul.

D. Destaque as ações de transição (representadas pelas setas) a medida que você passe por elas utilizando uma caneta verde. Por exemplo, o diagrama de estado fornecido no exemplo 5 contem sete ações de transição. A seta saindo do estado "authorizing" e voltando para este mesmo estado representa uma ação que não modifica oficialmente o estado do objeto.

E. Pense sobre os estados e ações que você identificou, e como podem estar representados em conjunto.

1) **Esteja certo que você pode entender e descrever o que esta acontecendo com o objeto apenas lendo a maquina de estado. Se você não pode, então a maquina de estado é ambígua. Indique isto no Formulário de Relato de Discrepância.**

V. **Encontre a classe ou hierarquia de classes, atributos e comportamentos na descrição de classes que correspondem aos conceitos do diagrama de estado.**

ENTRADAS: Descrição de Classes (CD);

Estados do Objeto (marcados em azul no SD);

Ações de Transição (marcados em verde no SD).

SAÍDAS: Atributos relevantes do objeto (marcado em azul no CD);
Comportamentos relevantes do objeto (marcado em verde no CD);
Relatório de Discrepâncias.

A. Utilize a descrição de classes para encontrar a classe ou hierarquia de classes que corresponde ao diagrama de estado.

1) **Se você não pode encontrar a classe correspondente preencha um relato de discrepância porque você encontrou uma inconsistência. A maquina de estado especifica uma classe que não esta descrita nas descrições de classes.**

B. Encontre como a classe responsável encapsula os estados marcados em azul descritos no diagrama de estados. Estados podem estar encapsulados por:

- 1 atributo explicitamente. (Um atributo existe e seus possíveis valores correspondem aos estados do sistema, por exemplo, atributo "modo" com estados "ligado" e "desligado").

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

- 1 atributo implicitamente. (Um objeto é considerado estar num estado específico dependendo do valor de algum atributo, mas o estado não é explicitamente gravado. Por exemplo, se a>5 o objeto se comporta de uma forma, para outros valores um outro comportamento é apropriado, mas nada explicitamente registra o estado corrente).
- Uma combinação de atributos.
- Tipo classe (Por exemplo, subclasses "fixed rate loan" e "variable rate loan" podem ser consideradas estados da classe pai "loan") Lembre-se de verificar a classe correspondente e todos os parentes na hierarquia de herança.
  Marque cada estado sublinhado em azul com um asterisco (*) quando ele for encontrado.

  1) **Se existem estados não marcados com asterisco então alguma coisa está faltando na descrição da classe. Se você pode identificar, a partir de seu conhecimento semântico do domínio, que estados extras não fazem sentido, então indique isto no Formulário de Relato de Discrepância, senão apenas indique que os dois diagramas estão inconsistentes.**

C. Para cada ação de transição marcada em verde no diagrama de estado, verifique se existem comport amentos de classe capazes de ativar aquela transição. Lembre-se de verificar na classe correntemente selecionada ou então nas classes mais altas na hierarquia de herança.

(Tenha em mente as seguintes exceções: 1) A transição depender de um atributo global, fora da hierarquia de classes. 2) Em instâncias de projeto de baixa qualidade, i.e. alto acoplamento e atributos públicos de classes, comportamentos em classes associadas podem modificar o valor de uma variável na classe diretamente)
Se a ação de transição é um evento (i.e. uma transição ocorre quando alguma coisa acontece) procure por um comportamento ou conjunto de comportamentos da classe que tratam este evento.

Se a ação de transição é uma restrição (i.e. uma transição ocorre quando alguma expressão se torna verdadeira ou falsa) procure por comportamento que podem trocar o valor de uma expressão de restrição. Por exemplo, observe as restrições "[payment ok]" e "[payment not ok]" no exemplo 5. Elas indicam quando as ações que elas descrevem podem acontecer, baseado na situação do pagamento.

Verifique as seguintes situações, e preencha o formulário de relato de discrepância se você encontrar alguma:

1) **Esteja certo que todas as ações são encapsuladas pela descrição da classe. Se elas não são, então alguma coisa está representada no diagrama de classes, mas não está na descrição de classes.**

2) **Tenha certeza que todas as restrições são encapsuladas pela descrição da classe. Se elas não são, então alguma coisa está representada no diagrama de estados, mas não está na descrição da classe.**

3) **Tenha certeza que os dados necessários para verificar uma restrição estão presentes na descrição da classe. Se eles não estão todos lá, então você encontrou uma informação no diagrama de estados que não está na descrição da classe.**

VI. Compare a descrição da classe com o diagrama de estados para ter certeza que a classe, como descrita, pode capturar a funcionalidade apropriada.

ENTRADAS: Estados do Objeto (marcados em azul no SD);
Ações de Transição (marcados em verde no SD).

SAÍDAS: Relatório de Discrepâncias

A. Considere a funcionalidade do sistema na qual a classe participa, como descrito pela descrição da classe, e os estados nos quais ela pode existir, como descrito pelo diagrama de estados.

1) **Utilizando seu conhecimento semântico sobre esta classe e os comportamentos que ela deveria encapsular, tenha certeza que todos os estados estão descritos. Se não, alguma coisa está faltando e a classe como descrita não pode se comportar como deveria. Indique isto no Formulário de Relato de Discrepância.**

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

Leitura 3– Diagramas de Seqüência x Estados

Objetivo: Verificar se toda transição de estado para um objeto pode ser realizada pelas mensagens enviadas e recebidas pelo objeto.

Entradas para o Processo:

5. Diagramas de Seqüência que descrevem as classes, objetos, e possíveis atores de um sistema e como eles colaboram para capturar os serviços do sistema.
6. Diagramas de Estados que descrevem os estados internos nos quais um objeto deve existir, e as possíveis transições entre estes estados.

Para **cada diagrama de estado**, execute os seguintes passos:

IV. **Leia o diagrama de estados para entender os possíveis estados de um objeto e as ações que disparam transições entre eles.**

ENTRADAS: Diagrama de Estados (SD).

SAÍDAS: Ações de Transição (marcadas e nomeadas em verde no SD);
Relatórios de Discrepância.

A. Determine qual classe está sendo modelada pelo diagrama de estado.
   1) **Se você não pode identificar a classe que está sendo modelada, então alguma coisa está sendo omitida ou esta ambígua. Indique isto no formulário de relato de discrepância.**

B. Acompanhe a seqüência de estados e as ações de transição (trocas do sistema durante o tempo de vida do objeto que provocam uma transição de um estado para outro) através do diagrama de estado. Comece pelo estado inicial (círculo cheio) e siga as transições ate que você encontre um estado final (círculo dobrado). Esteja certo que você passou por todas as transições.

C. Marque as ações de transição (representada pelas setas) a medida que você as encontre usando uma caneta verde. Por exemplo, o diagrama de estado fornecido no exemplo 5 contem sete ações de transição. A seta saindo do estado "authorizing" e voltando para este mesmo estado representa uma ação que não modifica oficialmente o estado do objeto. Dê a cada ação um único label [A1, A2, …]

D. Pense sobre os estados e ações que você identificou e como eles podem estar representados em conjunto.
   1) **Esteja certo que você pode entender e descrever o que esta acontecendo com o objeto apenas lendo a maquina de estado. Se você não pode, então a maquina de estado é ambígua. Indique isto no Formulário de Relato de Discrepância.**

V. **Leia os diagramas de seqüência para entender como as ações de transição são realizadas por mensagens que estão sendo enviadas e recebidas pelo objeto relevante.**

ENTRADAS: Diagrama de Estados (SD);
Ações de transição (marcadas e nomeadas em verde no SD);
Diagramas de Seqüência (SqD).

SAÍDAS: Mensagens dos Objetos (marcadas e nomeadas em verde no SqD);
Relatório de Discrepâncias.

A. Pegue os diagramas de seqüência e escolha aqueles que utilizam o objeto modelado pelo diagrama de estados; utilize este subconjunto de diagramas de seqüência para o restante desta atividade.
   1) **Se não existem diagramas de seqüência que utilizem esta classe, então preencha um relato de discrepância porque existe informação num diagrama de estado que não aparece nos diagramas de seqüência.**

Para cada diagrama de seqüência identificado previamente faça:

B. Leia o diagrama para identificar os serviços do sistema sendo descrito e as mensagens que este objeto recebe.

C. Pense sobre quais estados de objeto no diagrama de estado estão *semanticamente* relacionados aos serviços do sistema. Marque as transições de estado que levam a estes estados, e utilize este subconjunto para o restante desta atividade.

D. Mapeie as mensagens de objeto nos diagramas de seqüência para as transições de estado no diagrama de estado. Cada ação de transição deve mapear para uma mensagem ou uma seqüência de mensagens. Para fazer isto, você precisará pensar sobre a semântica associada às mensagens do sistema. Elas estão contribuindo para alcançar algum serviço maior do sistema ou então uma funcionalidade? Elas têm alguma relação com os tipos de estado que este objeto deveria estar? Quando você tiver feito o mapeamento, marque as mensagens relacionadas e as ações de transição com um asterisco (*). Nomeie as mensagens com o mesmo *label* dado para suas ações associadas do diagrama de estados.
   1) **Tenha semanticamente certeza que você pode fazer este mapeamento. Se não, então existem mensagens necessárias para a transição de estado que não estão no diagrama de**

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

seqüência. Preencha um relato de discrepância, porque informação incluída em um diagrama não esta incluída em outro.

E. Procure por restrições e condições nas mensagens que você acabou de mapear para as transições de estado. Um exemplo de restrição pode ser "t>0", isto é, se uma mensagem pode ou não ser enviada depende do valor de algum atributo t. Certifique que quaisquer restrição/condição encontrada seja capturada em algum lugar do diagrama de estados. Esta informação deveria ser capturada por: 1) informação de estado (i.e. o fato que t>0 corresponde a um estado particular do sistema), 2) informação de transição (i.e. alguma transição de estado ocorre quando t>0 torna-se verdadeira ou falsa) 3) nada (i.e. esta informação não é relevante ou importante para o diagrama de estados). Se qualquer um dos seguintes ocorrer, então preencha um relato de discrepância:

1) **Esteja certo que você pode encontrar uma correspondência entre condições e restrições nos diagramas de seqüência e estados. Se não, então um diagrama possui informação que não está no outro.**
2) **Para a informação que aparece em ambos os diagramas, certifique que está consistente. Se não está, então você encontrou que uma mesma informação está representada em dois diagramas diferentes de forma inconsistente.**

VI. Reveja os diagramas marcados para estar certo que todas ações de transições estão sendo levadas em consideração.

ENTRADAS: Ações de Transição (marcadas e nomeadas em verde no SD);
Mensagens de Objetos (marcadas e nomeadas em verde no SqD);.

SAÍDAS: Relatórios de Discrepância.

A. Reveja o diagrama de estados procurando por ações de transições não marcadas com asterisco e que poderiam não estar associada a mensagens de objetos.

1) **Se a ação de transição foi nomeada como uma restrição, veja se você pode encontrar uma mensagem ou seqüência de mensagens capaz de satisfazer a restrição. Se não, você encontrou informação representada em um diagrama, mas não representada em outro. O diagrama de estados requer serviços do sistema que não estão sendo descritos em nenhum dos diagramas de seqüência. Preencha um relato de discrepância.**
2) **Se a ação de transição foi marcada como um evento, veja se você pode encontrar uma mensagem, seqüência de mensagens, ou algum evento executado por um ator que provoca a ação de transição. Se não, você encontrou informação representada em um diagrama que não esta representada em outro. O diagrama de estados necessita de serviços que não estão representados em nenhum dos diagramas de seqüência. Preencha um relato de discrepância.**

B. Se as mensagens marcadas com asterisco e ações de transição identificadas no passo anterior aparecem no mesmo diagrama de seqüência, tenha certeza elas aparecem numa ordem lógica. Isto é, suponha que mensagens que tratam a ação A1 aparecem antes de mensagens que tratam a ação A2 em um diagrama de seqüência. Isto significa que A1 deve cronologicamente vir antes que A2. Então você deveria estar certo que A1 pode ser alcançado antes do que A2 também no diagrama de estados.

1) **Se a ordem não é compatível, então preencha um relato de discrepância, pois a informação está representada em dois diagramas, mas de forma inconsistente.**

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

Leitura 4– Diagrama de Classes x Descrições de Classes

Objetivo: Verificar se as descrições detalhadas das classes contêm toda a informação necessária e de acordo com o diagrama de classes, e se a descrição das classes possuem sentido semântico.

Entradas para o processo:

7. Um diagrama de classes (possivelmente dividido em pacotes) descrevendo as classes do sistema e como elas estão associadas.
8. Um conjunto de descrições de classes que lista as classes de um sistema juntamente com seus atributos e comportamentos.

III. **Leia o diagrama de classes para entender as propriedades necessárias das classes do sistema.**

ENTRADAS: Diagrama de Classes
Descrição de Classes.

SAÍDAS: Relatório de Discrepâncias.

Para **cada classe no diagrama de classes** execute os seguintes passos:

A. Encontre a descrição da classe correspondente. Marque a classe na descrição de classes com um asterisco azul quando a encontrar.

1) **Se você não pode encontrar a descrição, preencha o Formulário de Relato de Discrepância, porque uma classe presente no diagrama de classes não está presente nas descrições de classes.**

B. Verifique o nome e a descrição textual da classe para assegurar que elas fornecem uma descrição *significativa* da classe que você está considerando neste momento. Verifique também se a descrição está utilizando o nível adequado de abstração.

1) **Utilizando seu conhecimento, certifique que você pode entender o propósito desta classe na descrição de alto nível. Se não, a descrição deve estar muito ambígua para ser utilizada neste modelo de projeto. Preencha um relato de discrepância relatando este problema: *Conhecimento adicional necessário para compreensão*.**

C. Verifique que todos os atributos estão descritos em conjunto com seus tipos básicos.

1) **Tenha certeza que o mesmo conjunto de atributos está presente em ambos os documentos, ou seja, na descrição de classes e no diagrama de classes. Se não, preencha um relato de discrepância porque informação está presente num documento mas não está presente em outro.**

2) **Certifique que esta classe pode *significativamente* encapsular todos estes atributos, isto é, faz sentido existir estes atributos na descrição da classe e que os tipos básicos associados aos atributos são *viáveis* de acordo com a descrição do atributo. Se não, preencha o formulário de relato de discrepância indicando esta questão: *Conhecimento adicional necessário para compreensão.***

D. Verifique que todos os comportamentos e restrições estão descritos.

1) **Tenha certeza que o mesmo conjunto de comportamentos e restrições está presente em ambos os documentos, ou seja, da descrição da classe e do diagrama de classes, e que eles utilizam o mesmo estilo ou nível de granularidade (por exemplo, pseudocódigo) para descrever os comportamentos. Se não, então informação em um diagrama não está presente em outro, ou está inconsistente entre os dois.**

2) **Tenha certeza que esta classe pode *significativamente* encapsular todos estes comportamentos. Certifique que as restrições fazem sentido para esta classe. Certifique que os comportamentos podem executar suas tarefas utilizando os atributos que foram definidos (para esta ou alguma outra classe). Se não, preencha um relato de discrepância indicando: *Conhecimento adicional necessário para compreensão*.**

3) **Tenha certeza que as restrições são satisfeitas utilizando os atributos e comportamentos que foram definidos. Se não, você encontrou uma situação onde os comportamentos e restrições como definidos não podem ser satisfeitos utilizando os atributos e comportamentos que foram definidos. Indique isto no formulário de relato de discrepância como um problema: *Conhecimento adicional necessário para compreensão.* Explique o que ocorreu.**

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

4) **Tenha certeza que os comportamentos para esta classe não dependem excessivamente dos atributos de outras classes para executar suas funcionalidades. (Observe que você deve fazer um julgamento de valor sobre o que significa "dependência excessiva". Você deve comparar o número de referências existentes de outras classes para esta classe com o o que existe no resto do sistema, e considerar o tipo de funcionalidade endereçada para determinar se esta dependência é realmente necessária). Se eles dependem, então você possivelmente encontrou uma situação inadequada projeto. Preencha um relato de discrepância indicando esta situação.**

E. Se o diagrama de classes especifica algum mecanismo de herança para esta classe, verifique que eles estão corretamente descritos.

1) **Esteja certo que o relacionamento de herança está incluído na descrição da classe. Se não está, preencha um formulário de relato de discrepância. Informação no diagrama de classes não está na descrição da classe.**

2) **Utilize a hierarquia de classes para encontrar os pais desta classe. Verifique que, *semanticamente*, um <nome de classe> é do tipo <classe pai>, e que faz sentido ter esta classe neste ponto da hierarquia. Se não, você descobriu uma questão de estilo em potencial: a hierarquia não deveria ser definida desta maneira. Preencha um relato de discrepância descrevendo: *Informação adicional utilizada no projeto*.**

F. Verifique que todos os relacionamentos das classes (associação, agregação e composição) estão corretamente descritos com respeito às indicações de multiplicidade.

1) **Esteja certo que os papéis dos objetos estão capturados na descrição da classe, e que a notação gráfica correta e utilizada no diagrama de classes. Se você encontrar um problema, preencha um relato de discrepância indicando que informação está omitida em um diagrama, ou se a notação esta incorreta.**

2) ***Semanticamente*, certifique que os relacionamentos fazem sentido dado o papel e os objetos relacionados. Por exemplo, se um relacionamento de composição está envolvido, os objetos conectados realmente se parecem como uma estrutura "todo-parte"? Se eles não fazem sentido então você descobriu uma questão potencial de estilo: os relacionamentos não deveriam estar definidos desta maneira. Preencha um relato de discrepância descrevendo: *Informação adicional utilizada no projeto*.**

3) **Se as cardinalidades são importantes, certifique que elas estão descritas na descrição da classe. Dado que você entendeu o relacionamento, tenha certeza que as quantidades utilizadas para os objetos são suficientes. Se não, preencha um relato de discrepância porque informação em um diagrama não está presente em outro.**

4) **Certifique que existe algum atributo representando o relacionamento. Se não, preencha um relato de discrepância indicando que informação em um diagrama não está presente em outro.**

5) **Certifique que o relacionamento utiliza um tipo básico viável, ou estrutura de tipos básicos (se múltipla cardinalidade está envolvida). Se não, preencha um relato de discrepância indicando: *Informação adicional utilizada no projeto*.**

IV. Reveja as descrições de classes quanto a informação extra.

ENTRADAS: Descrição da Classe.

SAÍDAS: Relatos de Discrepância

A. Reveja as descrições de classes para ter certeza que todas as classes descritas oficialmente aparecem no diagrama de classes.

1) **Certifique que não existem classes sem asterisco na descrição de classes. Se existe alguma, preencha um relato de discrepância porque uma classe na descrição de classes não está presente no diagrama de classes.**

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

Formulário de Relato de Discrepância para Leitura Horizontal

Nome do Projeto: Equipe: __ Técnica de Leitura Horizontal: _____

Início da Inspeção: _____ (hora) Data: ___ (data)

Documentos que estão sendo lidos [preencha o nome e o tipo]:
Documento 1: _____ Documento 2: _____

Tipo de Conceito:

(AC) ator	(AT) atributo	(BE) comportamento	(CA) cardinalidade
(CO) condição	(CR) restrição	(DA) dado	(IN) herança
(ME) mensagem	(OB) objeto/classe	(RE) relacionamento	(RO) papel

Tipo de Discrepância (Tipo Disc.):
(1) presente no Documento 1 mas não no Documento 2
(2) presente no Documento 2 mas não no Documento 1
(3) presente em ambos os documentos, mas inconsistente ou ambíguo
(4) presente em ambos os documentos, mas usando uma representação ou notação incorreta
(5) presente em ambos os documentos, mas representa informação estranha
(6) faltando em ambos os documentos [explique abaixo]

Severidade (Sev.):
(NS) Não é sério. Mas precisa verificar este documento.
(IN) Esta discrepância invalida esta parte do documento. Verifique ambos os documentos.
(SE) Sério. Não é possível continuar a leitura deste documento. Ele deveria ser reorganizado.

Preencha a tabela com as discrepâncias que encontrou:

Disc.#	Tipo de Conceito	Nome	Tipo Disc.	Sev.	Comentários
01					
02					
03					
04					
05					
06					
07					
08					
09					
10					
11					
12					
13					
14					
15					

(Use outro lado se necessário)

Fim da Inspeção: _____ (hora)

Utilize a estrutura seguinte para detalhar algumas das discrepâncias encontradas que você considera importante descrever (todas as discrepâncias sérias e dos tipos 5 e 6 devem ser explicadas):
Número da Discrepância (o mesmo número usado na tabela): xx
Descrição:

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

Disc.#	Tipo do Conceito	Nome	Tipo Disc.	Sev.	Comentários
16					
17					
18					
19					
20					
21 22					
23					
24					
25					
26					
27					
28					
29					
30					
31					
32					
33					
34					
35					
36					
37					
38					
39					
40					
41					
42 43					
44					
45					
46					
47 48					
49					
50					
51					
52					
53					
54					
55					

(Utilize tabelas adicionais se necessário)

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

APPENDIX C.2 – OORTs 3.0 – Vertical Reading - Portuguese Version

Leitura 5– Descrições de Classes x Descrição de Requisitos

Objetivo: Verificar se os conceitos e serviços descritos pelos requisitos funcionais estão capturados apropriadamente pela descrição das classes.

Entradas para o Processo:

3. Um conjunto de requisitos funcionais que descrevem os conceitos e serviços que são necessários no sistema final.
4. Um conjunto de descrições de classes que lista as classes de um sistema juntamente com seus atributos e comportamentos.

IV. Leia a descrição de requisitos para entender a funcionalidade descrita.

ENTRADAS: Conjunto de Requisitos Funcionais (FR).

SAÍDAS: Classes/Atributos candidatos (marcados em azul nos FRs);
Serviços candidatos (marcados em verde nos FRs);
Restrições e condições para os serviços (marcados em amarelo nos FRs).

A. Leia totalmente cada um dos requisitos funcionais para entender a funcionalidade que ele escreve.

B. Encontre os substantivos na descrição do requisito; eles são candidatos a se tornar classes, objetos, ou atributos no projeto do sistema. Sublinhe os substantivos com uma caneta azul.

C. Encontre os verbos, ou descrições de ações, que são candidatos a serem serviços ou comportamentos no sistema. Sublinhe os verbos ou descrições de ações com uma caneta verde.

D. Procure por descrições de restrições ou condições nos substantivos e verbos que você identificou nos dois passos anteriores. Preste atenção especialmente aos requisitos não funcionais, que tipicamente contem restrições e condições relacionadas as funcionalidades do sistema. Por exemplo, examine se os relacionamentos entre os conceitos foram identificados. Pergunte se existem restrições explícitas ou limitações na forma que as ações são executadas. Tente observar se quantidades definidas foram especificadas em alguma parte do requisito (veja Exemplo 4). Sublinhe estas condições e restrições com uma caneta amarela.

V. Compare as descrições de classes aos requisitos para verificar se os requisitos foram capturados apropriadamente.

ENTRADAS: Conjunto de Requisitos Funcionais (FR);
Descrição das Classes (CD).

SAIDAS: Conceitos relacionados que foram marcados no FR e CD;
Relatórios de Discrepância.

A. Para cada descrição de ação sublinhada em verde nos requisitos funcionais, tente encontrar um comportamento associado ou combinação de comportamentos na descrição da classe. Utilize dicas sintáticas (i.e. nome do comportamento que é similar ou sinônimo para uma descrição de ação) para ajudar você na busca, mas certifique que o significado *semântico* da função nos requisitos e projeto de alto nível é o mesmo. Quando encontrado, marque ambos o nome do comportamento(s) na descrição da classe e a descrição da atividade nos requisitos com um símbolo (*) verde.

   1) **Esteja certo que as classes recebem a informação correta para desempenhar os comportamentos requisitados. Certifique que resultados *viáveis* são produzidos. Se não, as classes não podem implementar a funcionalidade de forma apropriada. Indique isto no formulário de relato de discrepância e marque se isto é provocado por funcionalidade omitida ou incorreta ou então informação ambígua.**

B. Para cada substantivo sublinhado em azul nos requisitos funcionais, tente encontrar uma classe associada na descrição de classes. Uma classe associada pode estar nomeada depois de um conceito dos requisitos, pode descrever uma classe geral dos quais o conceito e uma instancia particular (i.e. um objeto), ou pode conter o conceito como um atributo. Utilize dicas sintáticas (e.g. um nome de classe que e similar ao nome de um conceito) para ajudar você na pesquisa, mas esteja certo que o significado *semântico* dos conceitos nos requisitos e projeto e o mesmo.

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

C. Se o conceito nos requisitos funcionais corresponde a um nome de classe na descrição de classes, marque ambos o nome da classe na descrição de classe e o conceito na descrição de requisitos com um símbolo azul (*).

1) **Tenha certeza que as descrições de classes contem informação suficiente relacionada aos conceitos que executam algum papel nesta funcionalidade e os nomes das classes tem alguma conexão com os substantivos que você marcou. Se não, ou se as classes estão utilizando informação ambígua para descrever os conceitos indique isto no formulário de relato de discrepância.**

2) **Tenha certeza que estas classes encapsulam (marcada em azul) atributos referentes aos substantivos que você marcou e os comportamentos (marcados em verde) relacionados com os verbos ou descrição de ações que você marcou. Certifique também que todas as restrições e condições identificadas para esta classe e referente a este requisito estão descritas. Se não, você encontrou informação importante dos requisitos omitida do projeto. Indique isto no formulário de relato de discrepância.**

D. Se o conceito nos requisitos funcionais corresponde a um atributo na descrição da classe, marque ambos o nome do atributo na descrição da classe e o conceito na descrição de requisitos com um símbolo azul (*).

1) **Tenha certeza que a descrição da classe esta utilizando tipos *viáveis* para representar a informação, dado que a descrição dos requisitos e as restrições e condições (marcadas em amarelo) sob os atributos foram observadas em sua definição. Se não, você encontrou informação incorreta no projeto. Indique isto no formulário de relato de discrepância.**

VI. Reveja a descrição da classe e os requisitos funcionais para certificar que todos os conceitos apropriados possuem correspondência entre os documentos.

ENTRADAS: Conjunto de Requisitos Funcionais (FR);
Descrição das Classes (CD).

SAÍDAS : Relatório de Discrepância.

A. Procure por descrições de funcionalidade nos requisitos que tenham sido omitidas do projeto.

1) **Certifique que não existem substantivos ou verbos não marcados com asterisco (*) nos requisitos Se existe pelo menos um, verifique se ele deveria ser incluído no projeto ou não esta descrito apenas para melhorar a legibilidade dos requisitos. Se ele deveria estar no projeto, então informação foi omitida do projeto. Indique isto no formulário para relato de discrepância.**

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

Leitura 6– Diagramas de Seqüência x Casos de Uso

Objetivo: Verificar que os diagramas de seqüência descrevem uma combinação apropriada de objetos e mensagens que trabalham em conjunto para capturar a funcionalidade descrita pelo caso de uso.

Entradas para o processo:

4. Um caso de uso que descreve conceitos importantes do sistema (os quais podem eventualmente ser representado como objetos, classes ou atributos) e os serviços que ele fornece.
5. Um ou mais diagramas de seqüência que descreve os objetos de um sistema e os serviços que ele fornece. Podem existir múltiplos diagramas de seqüência para um dado caso de uso desde que um caso de uso tipicamente descreverá múltiplos "caminhos de execução" através da funcionalidade do sistema. O conjunto correto de diagramas de seqüência para um caso de uso deve ser selecionado utilizando informação de rastreabilidade, ou por alguém que tenha conhecimento semântico sobre o sistema. Encontrar o conjunto correto de diagramas de seqüência sem informação de rastreabilidade ou conhecimento sobre o sistema será difícil.
6. As descrições de classes de todas as classes do diagrama de seqüência.

IV. Identifique a funcionalidade descrita por um caso de uso, e os conceitos importantes do sistema que são necessários para realizar a funcionalidade.

ENTRADAS: Caso de Uso (UC)

SAÍDAS: Conceitos do Sistema (marcados em azul no UC);
Serviços fornecidos pelo sistema (marcados em verde no UC);
Dados necessários para realização dos serviços (marcados em amarelo no UC).

A. Leia o caso de uso para entender a funcionalidade que ele descreve.
B. Encontre os substantivos incluídos no caso de uso; eles descrevem os conceitos do sistema. Sublinhe e numere cada substantivo único com uma caneta azul assim que ele for encontrado. (Isto é, se um substantivo em particular aparece várias vezes, marque com o mesmo número cada vez que o encontrar).
C. Para cada substantivo identifique os verbos que descrevem ações aplicadas *nos* ou *pelos* substantivos. Sublinhe os serviços identificados e os numere (na ordem em que devem ser executados) com uma caneta verde. Procure por restrições e condições que são necessárias para que este conjunto de ações seja executado. Como um exemplo, considere o Exemplo 1, no qual restrições e condições foram marcados. Neste caso de uso, existe um exemplo de uma restrição ("*The Customer can only wait for 30 seconds for the authorization process*") e uma condição ("*time of payment is the same as the purchase time*").
D. Identifique também qualquer informação ou dado que é requisitado para ser enviado ou recebido de forma a executar as ações. Marque os dados em amarelo como "Di,j" onde os subscritos i e j são os números associados aos nomes entre os quais a informação é trocada.

V. Identificar e inspecionar os diagramas de seqüência relacionados, para identificar se a funcionalidade correspondente está precisamente descrita e se os comportamentos e dados estão representados na ordem correta.

ENTRADAS: Caso de Uso, com conceitos, serviços e dados marcados;
Diagrama de Seqüência (SD).

SAÍDAS: Conceitos do Sistema (marcados em azul no SD);
Serviços fornecidos pelo sistema (marcados em verde no SD);
Dados trocados entre objetos (marcados em amarelo no SD).

A. Para cada diagrama de seqüência, sublinhe os objetos do sistema com uma caneta azul. Numere os com os números correspondentes dos casos de uso.
B. Identifique os *serviços* descritos pelos diagramas de seqüência. Para fazer isto, você precisará examinar a informação trocada entre os objetos e classes nos diagramas de seqüência (as setas horizontais). Se a informação trocada está muito detalhada, no nível de mensagens, você precisará abstrair várias mensagens em conjunto para entender os serviços que elas fornecem. Sublinhe os serviços identificados e os numere (na ordem em que ocorrem no diagrama) com uma caneta verde. Procure pelas condições que permitem a ativação das ações.
C. Identifique a informação (ou dado) que é trocado entre as classes do sistema. Marque o dado em amarelo como "Di,j" onde os subscritos i e j são os números associados aos objetos entre os quais a informação é trocada.

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

VI. **Compare os diagramas marcados para determinar se eles representam os mesmo conceitos do domínio.**

ENTRADAS: Caso de Uso, com conceitos, serviços e dados marcados;
Diagrama de Seqüência, com objetos, serviços e dados marcados.

SAÍDAS: Relatório de Discrepâncias.

A. Para casa substantivo marcado em azul no caso de uso, procure no diagrama de seqüência para ver se o mesmo substantivo está representado. Marque o substantivo no caso de uso e no diagrama de seqüência com um asterisco azul (*) se ele pode ser encontrado no diagrama de seqüência.
    1) **Se existe algum substantivo sem asterisco no caso de uso, isto significa que um conceito foi usado para descrever funcionalidade no caso de uso mas não foi representado no diagrama de seqüência. Para cada um dos substantivos no diagrama de seqüência, encontre a classe correspondente na descrição de classes e verifique se o substantivo não marcado com asterisco é um atributo. Se o substantivo não marcado com asterisco não aparece como um atributo de nenhuma das classes, você encontrou uma omissão. Um conceito foi descrito no caso de uso, mas não apareceu no projeto do sistema. Preencha um relatório de discrepância porque funcionalidade necessária foi omitida.**
    2) **Se existem substantivos não marcados com asterisco no diagrama de seqüência você encontrou um substantivo estranho, ou um substantivo descrevendo um conceito de baixo nível no diagrama de seqüência. Pense se este conceito é necessário para o projeto de alto nível, e se ele representa um nível de detalhe que é apropriado neste momento. Se não é apropriado, preencha um relato de discrepância porque esta informação é estranha.**

B. Identifique os serviços descritos pelo diagrama de seqüência, e compare os com a descrição usada no caso de uso. As classes/objetos estão trocando mensagens na mesma ordem especificada no caso de uso? Os dados que aparecem nas mensagens dos diagramas de seqüência foram corretamente descritos nos casos de uso? É possível para você entender a funcionalidade esperada apenas lendo o diagrama de seqüência?
    1) **Esteja certo que as classes trocam mensagens na mesma ordem especificada. Se não, pense se isto representa um defeito. Usualmente, trocando a ordem da mensagem terá efeito sobre a funcionalidade. Mas algumas vezes mensagens podem ser trocadas sem afetar a saída; outras vezes, mensagens podem ser executadas em paralelo, ou condições podem assegurar que somente uma ou outra mensagem é executada de qualquer forma. Se trocando a ordem trocará a funcionalidade, preencha um relatório de discrepância porque a informação do projeto está incorreta.**
    2) **Esteja certo que os dados trocados estão todos na mensagem correta e que as mensagens vão para as classes corretas (i.e. as marcações "Di,j" para os dados são as mesmas entre os diagramas) Esteja certo que as mensagens fazem sentido para os objetos que as recebem e enviam, e para ativar os serviços relevantes. Se não, isto significa que o diagrama de seqüência está usando informação incorretamente. Preencha um relatório de discrepância descrevendo o problema.**

C. Todas as restrições e condições do caso de uso podem ser observadas neste diagrama de seqüência? Existe algum detalhe do caso de uso faltando aqui?
    1) **Esteja certo que as restrições são observadas. Esteja certo que todos os comportamentos e dados relativos ao diagrama de seqüência são relacionados ao caso de uso. Se não, isto significa que o diagrama de seqüência esta usando informação incorretamente. Preencha um relatório de discrepância descrevendo o problema.**

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

Leitura 7 – Diagramas de Estados x Descrição de Requisitos e Casos de Uso

Objetivo: Verificar se os diagramas de estado descrevem apropriadamente os estados dos objetos e eventos que disparam as trocas de estado conforme descritos pelos requisitos e casos de uso.

Entradas para o processo:
4. O conjunto de todos os diagramas de estados, cada descrevendo um objeto do sistema.
5. O conjunto de requisitos funcionais que descreve os conceitos e serviços necessários ao sistema final.
6. O conjunto de casos de uso descrevendo os conceitos importantes do sistema.

Para **cada diagrama de estados**, execute os seguintes passos:

V. **Leia o diagrama de estados para entender basicamente o objeto que ele está modelando.**

VI. **Leia a descrição de requisitos para determinar os possíveis estados do objeto, que estados são adjacentes entre si, e eventos que causam as trocas de estado.**

ENTRADAS: Diagramas de Estados (SD)
Descrição de Requisitos (RD)

SAÍDAS: Estados dos Objetos (marcados em azul no SD)
Matriz de Adjacência

A. Pegue o diagrama de estados e apague qualquer asterisco (*) daqueles existentes e resultantes de interações anteriores desta etapa. Agora, leia os requisitos procurando por locais onde o conceito está descrito ou por algum requisito funcional nos quais o conceito participa ou é afetado. Quando você localizar algum requisito, marque a lápis, para facilitar sua utilização pelo estante desta etapa, com um asterisco (*). Mantenha o foco nestas partes da Descrição de Requisitos (RD) pelo resto desta etapa.
B. Localize descrições para todos os estados diferentes que este objeto pode estar. Para localizar um estado, procure por valores de atributos ou combinações de valores de atributos que possam modificar o comportamento do objeto. Quando localizar um estado sublinhe-o com uma caneta azul e associe um numero a ele.
C. Agora identifique qual destes estados numerados é o estado inicial. Utilizando uma caneta azul, marque-o com um "I". Da mesma forma, marque o estado final com um "E".
D. Quando você tiver encontrado todos os estados e utilizando uma folha de papel separada, crie uma matriz numerada de 1..N na primeira linha e 1..N na primeira coluna, onde 1..N representa os números que você associou aos estados identificados no passo anterior desta etapa.
E. Para cada par de estados, verifique se o objeto pode trocar do estado representado pelo número da esquerda para o estado representado pelo número da primeira linha. Então marque a célula na interseção da linha e da coluna. Se você pode determinar o evento(s) que causa a troca de estado coloque-o na célula, se não apenas coloque uma marca na célula (o evento será determinado em próxima iteração). Se você pode determinar que não é possível acontecer a transição então coloque um X na célula. Se você não pode identificar definitivamente então deixe a célula em branco por enquanto.
F. Para qualquer evento que tenha identificado acima, se existem quaisquer restrições descritas nos requisitos, então as escreva junto ao evento na matriz.

VII. **Leia os Casos de Uso e determine os eventos que podem causar trocas de estado.**

ENTRADA: Casos de Uso

SAÍDA: Matriz de Adjacência Completa

A. Leia os dos casos de uso e encontre aqueles nos quais o objeto participa. Mantenha o foco nestes casos de uso pelo resto desta etapa.
B. Para cada célula marcada na matriz de adjacência, procure pelos casos de uso e determine que evento(s) pode causar a transição. Estes eventos podem não ser óbvio e podem obrigar que você abstraia o caso de uso para pensar sobre o que está atualmente acontecendo com cada objeto. Apague a marca da célula e escreva o nome deste evento(s) no seu lugar.
C. Para cada célula em branco na matriz de adjacências, veja se algum evento que pode causar a transição está descrito nos casos de uso. Se estiver, então escreva este evento na célula, se não então coloque um X na célula.

VIII. **Leia o diagrama de estados para determinar se os estados descritos estão consistentes com os requisitos e se as transições estão consistentes com os requisitos e casos de uso.**

ENTRADA: Descrição de Requisitos;

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

Diagrama de Estados (SD);
Matriz de Adjacência (AM).

SAÍDA: Relatórios de Discrepância

A. Para cada estado marcado e numerado na descrição de requisitos, encontre o estado correspondente no diagrama de estados e utilizando uma caneta azul, marque-o com o mesmo número utilizado nos requisitos. Preste atenção, pois o mesmo estado pode ter um nome diferente nos requisitos daquele que esta no diagrama de estados. Para identificar se dois nomes diferentes representam o mesmo estado, você deve utilizar sua compreensão da descrição do estado oriunda dos requisitos e a informação contida no diagrama de estado. Isto pode ser um processo interativo, pois se estados parecem estar faltando, você deve voltar e procurar de novo no que você identificou e ter certeza que está correto. Se você encontrou qualquer problema, preencha um relato de discrepância.

1) **Esteja certo que você pode encontrar todos os estados. Se um estado está faltando, procure ver se dois ou mais estados que você marcou nos requisitos foram combinados em um estado no diagrama de estados. Se não, então informação foi omitida do projeto. Se sim, então esteja certo que esta combinação faz sentido. Se não faz, então o projeto possui informação incorreta.**

2) **Tenha certeza que não existem estados adicionais no diagrama de estados. Procure ver se um estado que você marcou nos requisitos não foi dividido em dois ou mais estados no diagrama de estados. Se não, então informação no projeto e estranha. Se sim, tenha certeza que esta divisão faz sentido. Se não faz então o projeto possui informação incorreta.**

B. Uma vez que você tem todos os estados marcados com números, utilizando a Matriz de Adjacência (AM), compare os eventos de transição na matriz com aqueles no SD. Para qualquer célula na AM marcada com um evento, verifique os estados correspondentes no SD para ter certeza que eles têm um evento para a transição entre eles, e assegure que o evento seja o mesmo.

1) **Certifique que todos os eventos na AM aparecem no SD. Se não, informação foi omitida do projeto. Se existem eventos extras no diagrama de estados, então o projeto possui informação estranha.**

C. Para cada restrição que foi marcada na AM, encontre-a no SD.

1) **Tenha certeza que você pode encontrar todas as restrições que estão no AM. Se você não pode, então informação foi omitida do projeto. Se existem restrições extras no diagrama de estados, então o projeto possui informações estranhas.**

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

Formulário de Relato de Discrepância para Leitura Vertical

Nome do Projeto: Equipe: __ Técnica de Leitura Vertical: _____

Início da Inspeção: _____ (hora) Data: ___ (data)

Documentos que estão sendo lidos [preencha o nome e o tipo. Documento 2 somente executando leitura 7]:
Documento 1: _____ Documento 2: _____

Tipo de Conceito:

(AC) ator	(AT) atributo	(BE) comportamento	(CA) cardinalidade
(CO) condição	(CR) restrição	(DA) dado	(IN) herança
(ME) mensagem	(OB) objeto/classe	(RE) relacionamento	(RO) papel

Discrepancy type (Disc. Type):
(1) funcionalidade ou conceito necessário foi omitido
(2) o projeto está incorreto em relação aos requisitos
(3) como o projeto implementa estes requisitos é ambíguo ou não completamente especificado
(4) a informação de projeto é estranha, i.e. não mencionada pelos requisitos.
(5) outro problema de projeto [explique abaixo]

Severidade (Sev.):
(NS) Não é sério. Mas precisa verificar este documento.
(IN) Esta discrepância invalida esta parte do documento. Verifique ambos os documentos.
(SE) Sério. Não é possível continuar a leitura deste documento. Ele deveria ser reorganizado.

Preencha a tabela com as discrepâncias encontradas. Descreva a funcionalidade dos requisitos, utilizando os números dos requisitos e números de página se possível:

Disc. #	Tipo do Conceito	Nome	Tipo Disc.	Identificação Requisito	Sev.	Comentários
01						
02						
03						
04						
05						
06						
07						
08						
09						
10						
11						
12						
13						
14						
15						

(Utilize a parte de trás se necessário)

Fim da Inspeção: _____ (hora)

Utilize a estrutura seguinte para detalhar algumas das discrepâncias encontradas que você considera importante descrever (todas as discrepâncias sérias e do tipo 5 devem ser explicadas):
Número da Discrepância (o mesmo número usado na tabela): xx
Descrição:

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland

Disc. #	Tipo do Conceito	Nome	Tpo Disc.	Identificação do Requisito	Sev.	Comentários
16						
17						
18						
19						
20						
21						
22						
23						
24						
25						
26						
27						
28						
29						
30						
31						
32						
33						
34						
35						
36						
37						
38						
39						
40						
41						
42						
43						
44						
45						
46						
47						
48						
49						
50						
51						
52						
53						
54						
55						
56						
57						
58						
59						
60						

(Utilize Tabelas Adicionais se precisar)

Copyright © 2002 COPPE/UFRJ-Brazil, Fraunhofer Center-Maryland, University of Maryland