COS603

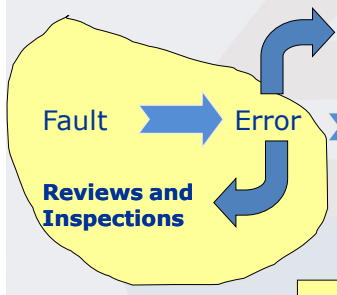
Software Validation, Verification and Testing

Guilherme H. Travassos
ght@cos.ufrj.br

Topic 2

Software Defects, Inspection Method and Techniques: ad-hoc, checklists and scenario-based reading techniques. Software Requirements Inspection

Verification, Validation and Testing



Defects Identification Techniques Efficiency

Technique	Efficiency
Unit Testing	15% a 50%
Integration Testing	25% a 40%
System Testing	25% a 55%
Beta Test (< 10 customers)	24% a 40%
Beta Test (> 1000 customers)	60% a 85%

Adapted from Capers Jones, Software defect-removal efficiency, IEEE Computer, March 1996

Evidence on Software Inspections
(academia)

Inspections significantly increase productivity, quality, and project stability.

Fagan´s law

Effectiveness of Inspections is fairly independent of its organizational form.

Porter-Votta's law

Perspective-based inspections are (highly) effective and efficient.

Basili´s law

A combination of different V&V methods outperforms any single method alone.

Hetzel-Myers law

Endres, A; Rombach, D. (2003). A Handbook of Software and Systems Engineering: Empirical Observations, Laws and Theories. Fraunhofer IESE Series on Software Engineering. Pearson/Addison Wesley.ISBN 0321154207

Evidence on Software Inspections
(academia)

- Quality entails productivity.
  - Mills-Jones hypothesis
- Error prevention is better than error removal.
  - May's hypothesis
- Proving of programs solves the problems of correctness, documentation, and compatibility.
  - Hoare's hypothesis
- Approximately 80 percent of defects come from 20 percent of modules.
  - Pareto–Zipf-type laws

Endres, A; Rombach, D. (2003). A Handbook of Software and Systems Engineering: Empirical Observations, Laws and Theories. Fraunhofer IESE Series on Software Engineering. Pearson/Addison Wesley.ISBN 0321154207

Evidence on Software Inspections
(industry)

Company	Software Category	Inspected Artifact	Results
AT&T	Telecom	Requirements, design, code and testing	Inspection has increased productivity and quality by 14%, being 20x more efficient than testing.
HP	Varied	Design, code, testing, documentation	An audit revealed an ineffective inspection process. Problems under discussion.
		Code	2 defects detected per hour. It is unlikely that 80% of defects could be caught by testing.
BRN	Telecom	Code	1 defect detected per hour. The process was 20x more efficient than testing.
Bull HN Information Systems	Operating system	Requirements, design, code, testing, documentation.	4 people's teams were twice as efficient as the one composed of 3.

Travassos, G.H. (2014). Software Defects: Stay Away from them. Do Inspections!. QUATIC 2014. Keynote. (in press)

Evidence on Software Inspections
(industry)

Company	Software Category	Inspected Artifact	Results
IBM	Operating system	Design and code	23% increasing in code productivity and 38% reduction of defects found in test stage.
ICL	Operating system	Design	40% to 50% increasing in defect detection. 1.2 hours per defect in inspection compared to 8.4 hours with testing.
JPL	Space system	Requirements, design, code, testing	0.5 hours to find defects versus 5 hours for other techniques.
MEL	Varied	Design, code	ROI calculated at 8:1. In 75 inspections the result was 7000 hours saved.
Shell Research	Geophysical software	Requirements	1 defect found every 3 minutes. Return on investment calculated at 30:1.

Travassos, G.H. (2014). Software Defects: Stay Away from them. Do Inspections!. QUATIC 2014. Keynote. (in press)

Benefits of inspections: early software defect detection

- Inspection results from Alaska SAR, JPL, where inspections were conducted in requirements and design phases (Miller, 1990):

	Req. Defects	Design Defects	Other Defects	Total
Req. Phase	520			520
Design Phase	22	513		535
Subsystem Test Phase			409	409
System Test Phase			4	4
Total	542	513	413	1468

Benefits of inspections: productivity and cost

Inspections improve productivity by finding defects when they are cheaper to discover and to correct:

Hours to *find* a defect (JSC):

Early in project	1.2 (easy), 1.4 (hard)
Late in project	1.5 (easy), 3 (hard)

Hours to *fix* a defect (JPL):

Inspections	0.7
Testing	5 to 18

Quantitative benefits of inspections

Example: Shuttle Software
(Primary Avionics Software Systems, PASS, JPL)

Inspections applied from 1982 to 1985 on requirements, design, code, test plans, specifications, procedures.

During this time operational defect rate was reduced from:

2.25 to 0.08 defects/KSLOC

One of the best examples of quality improvement from inspections!

Qualitative benefits of inspections: useful spin-off for participants

➢Learning by experience

- Participants learn about rationales and patterns in defect discovery
- Participants learn about good development patterns

In the long term...

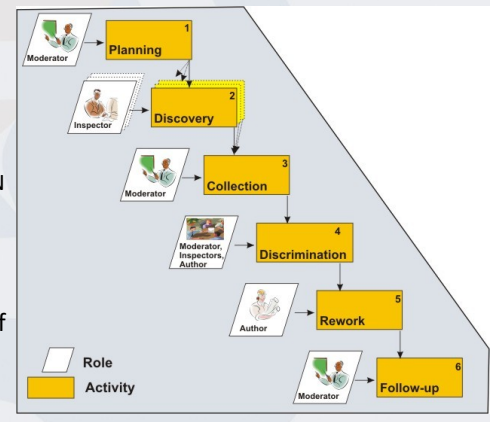
- inspections can convince participants to develop more understandable and easier to maintain products

Inspections help to integrate defect prevention processes into the development processes!

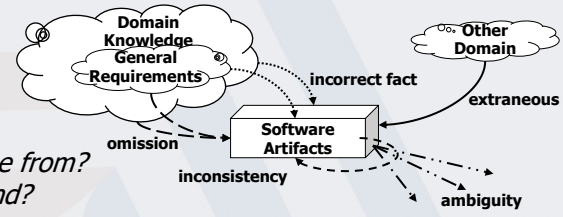
Software Inspections

➢Software Inspection Process Reorganization
(SAUER et al., 2000)

– Asynchronous meetings and geographically distributed teams.
  - Cost and scheduling conflicts can be reduced. Same effectiveness. VOTTA (1993), JOHNSON (1998)
  - Experimental Studies: (JOHNSON e TJAHJONO, 1997), (PORTER e JOHNSON, 1997), (JOHNSON e TJAHJONO, 1998).
– Introduces changes to reduce costs and time for the accomplishment of this type of inspection.



Software Defects



From where defects come from?
What types of defects we can find?

Defect	***General Description***
Omission	Necessary information about the system has been omitted from the software artifact.
Incorrect Fact	Some information in the software artifact contradicts information in the requirements document or the general domain knowledge.
Inconsistency	Information within one part of the software artifact is inconsistent with other information in the software artifact.
Ambiguity	Information within the software artifact is ambiguous, i.e. any of a number of interpretations may be derived that should not be the prerogative of the developer doing the implementation.
Extraneous Information	Information is provided that is not needed or used.

Travassos, G. H., Shull, F. and Carver, J. Working with UML: A Software Design Process Based on Inspections for the Unified Modeling Language, in Advances in Computers, vol. 54, Academic Press, 2001

Software Inspection Techniques

Question: How does one detect defects in software artifacts?

Answer:

1 By reading the document.

2 By understanding what the document describes.

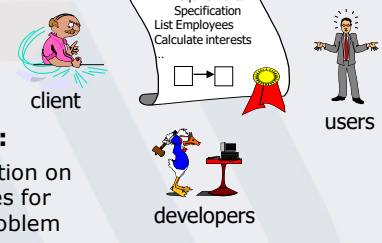
3 By checking the required quality properties.

Software Requirements

- Requirements
  - A requirement is (IEEE Software Engineering Standards, 1987):
    (1) a condition or capability needed by a user to solve a problem or achieve an objective
    (2) a condition or capability that must be met or possessed by a system…to satisfy a contract, standard, specification, or other formally imposed document…
  - The document describing all the software requirements (usually using an understandable customer format) is the Requirements Description.
  - The document specifying these requirements, in a format better suited for implementation, is the Requirements Specification
  - Generally, both requirements description and specification describe what the proposed software should do without describing how the software shall do

Software Requirements

- **Functional requirements:**
  - describes an interaction between the system and its environment
- **Non-functional requirements:**
  - or constraint, describes a restriction on the system that limits our choices for constructing a solution to the problem



- A Requirements Specification is important because
  - it establishes the agreement basis between the client and supplier on what the software product will do
  - it provides a reference for the final product validation
  - having high quality requirements specification is a prerequisite to high-quality software
  - it reduces the development cost by avoiding "behaviors guessing"

Software Requirements Defects

➢ Types of defects usually found in requirements specifications

DEFECT TYPE	PERCENTAGE (%)
Incorrect Fact	40
Omission	31
Inconsistency	13
Ambiguity	5
Incorrect Location	2

Adapted from DAVIS, A., 1990, "Software Requirements Analysis and Specification", Prentice Hall, Englewood Cliffs, NJ.

Software Defects

- Example requirements: Gas Station Control System (GSCS)
  - **Overview:**
    - "... The gas station allows customers to purchase gas (self-service) or to pay for maintenance work done on their cars. Some local businesses may have billing accounts set up so that the business is sent a monthly bill, rather than paying for each transaction at the time of purchase. There will always be a cashier on-duty at the gas station to accept cash payments or perform system maintenance, as necessary."
    - The requirements in this excerpt "...concern how the system receives payment from the customer. A customer has the option to be billed automatically at the time of purchase, or to be sent a monthly bill and pay at that time. Customers can always pay via cash or credit card. "

Travassos, G. H., Shull, F. and Carver, J. Working with UML: A Software Design Process Based on Inspections for the Unified Modeling Language, in Advances in Computers, vol. 54, Academic Press, 2001

Software Defects

- **Functional Requirement 5**
  - If payment is to be made by cash, the cashier is responsible for accepting the customer's payment and making change, if necessary. When payment is complete, the cashier indicates this on the cashier's interface. The GSCS and the gas pump interface then return to the initial state.

What is the purchase price?

- Information was lost during the creation of the requirements.
  - To handle a cash transaction, the cashier must know what the purchase price was.
  - This information has been left out of the description of the functionality - therefore we have a defect!

Software Defects

- **Functional Requirement 3**
  - If the customer has selected to pay at the time of purchase, he or she can choose to pay by cash or credit card. If the customer selects cash, the gas pump interface instructs the customer to see the cashier. If the customer selects credit card, the gas pump interface instructs the customer to swipe his or her credit card through the credit card reader. If an invalid or no selection is made, the GSCS will use the credit card payment option, which is the default.

Is this the correct response?

- Information was translated incorrectly
  - In the example, domain knowledge should indicate that defaulting to credit card payment is an incorrect response. (What kind of transaction ever happens this way?)
  - Because we know that this functionality should not be implemented the way it is described, we have a defect.

Software Defects

- **Functional Requirement 2**
  - After the purchase of gasoline, the gas pump reports the dollar amount of the purchase to the GSCS. The maximum value of a purchase is $999.99. The GSCS then causes the gas pump interface to query the customer as to payment type.

...

- **Functional Requirement 100** ?
  - If payment is to be made by credit card, then the card reader sends the account number to the GSCS. If the GSCS receives an invalid card number, than a message is sent to the gas pump interface asking the customer to swipe the card through the card reader again. After the account number is obtained, the account number and purchase price are sent to the credit card system, and the GSCS and gas pump interface are reset to their initial state. The purchase price sent can be up to $10000.

- Information was described inconsistently
  - Because we don't know from domain knowledge which of the two descriptions is correct, we have found a defect.

Software Defects

- **Functional Requirement 6.1**
  - The customer must give the billing account number to the cashier, who then enters it at the cashier's interface. If a valid billing account number is entered, then the billing account number, purchase price, and a brief description of the type of transaction is logged. If an invalid billing account number is entered, an error message is displayed and the cashier is prompted to enter it again. The cashier must also have the option to cancel the operation, in which case the cashier's interface reverts to showing the purchase price and the cashier can again either receive cash or indicate that monthly billing should be used.

- Information could be translated incorrectly
  - What information is stored?
  - What does "transaction type" mean?
    - Gas / Maintenance?
    - Full / Partial?
    - Credit card / Cash / Monthly bill?

Software Defects

- **Functional Requirement 7**
  - To pay a monthly bill, the customer must send the payment along with the billing account number. The cashier enters monthly payments by first selecting the appropriate option from the cashier's interface. The GSCS then sends a message to the cashier's interface prompting the cashier to enter the billing account number, the amount remitted, and the type of payment. If any of these pieces of information are not entered or are invalid, payment cannot be processed; an error message will be displayed, and the cashier's interface will be returned to the previous screen. If the type of payment is credit card, the credit card account number must also be entered, and then the paper credit card receipt will be photocopied and stored with the rest of the year's receipts.

Outside scope of system

- Extraneous Information Introduced
  - Information that may itself be correct information causes a problem because it should not be part of the system.
  - How could we test for this?

Reading techniques

Problem ▶ Inspectors often do not know how to read the document!

Reason

- **Developers learn how to write requirements, design, code, etc. But, they do not learn how to read them.**
- **Little technical support for the reading process is currently available.**

Solution

- **Provide well-defined and tailored reading techniques.**

Benefit

- **Increase the cost-effectiveness of inspections**
- **Provide models for writing documents of higher quality**
- **Reduce human influence on inspection results (i.e., ensure a more engineering approach in inspections)**

Inspection Technique: *ad-hoc*

Inspector reads the document accordingly its own perspective and knowledge

Individual experience affects the final results:

- Focus on the inspector expertise
- Individual productivity
- Hard to guarantee the inspector read the document in the correct way because each inspector applies its own review approach

There is no document coverage guarantee

Cost/efficiency (#defects/time of inspection) tend to be better when inspectors have high experience (> inspection cost)

Inspection Technique: checklist

Inspector must follow a list of items representing the software characteristics although following an ad hoc approach (checklists describe what to look for, but not how to look for)

More directed final result:

- Quality characteristics defined *a priori*
- Individual productivity
- Hard to guarantee the inspector reads the document in the correct way even defining the quality characteristics to be reviewed, because each inspector applies its own review approach

Document coverage concerned with the checklist items and inspector approach

Cost/efficiency depends on the checklist and inspectors

Checklist can be tailored or specifically built to capture a specific quality characteristic

Inspection Technique: checklist

Example: Design Completeness

Inspection Questions	Yes (Pass)	No (Fail)
Package Designs: Does the SDD document all significant package design decisions?		
Unit Designs: Does the SDD document all significant unit design decisions?		
Thoroughly Documented: Are design decisions for the current release documented as completely and as thoroughly as is known at the present time? Note that information relevant to future releases need not be completely documented.		
Current TBDs: Is the acronym "TBD" used to signify that the associated design decisions have not yet been determined and documented?		
No TBDs at Release: Does the final SDD for a release not contain any "TBDs" for that release?		

Software Design Document (SDD) Inspection Checklist – OPEN Process Framework
http://www.opfro.org/index.html?Components/WorkProducts/DesignSet/SoftwareDesignDocument/SoftwareDesignDocumentInspectionChecklist.html~Contents

Inspection Technique: checklist

Defect Report form
Name: J.J. XPT
Used Checklist: 01
Reviewed Document: Specification Requirements for the USE CASE Tool to support PBR.
Inspection time: 2 hs

Defect No.	Page No.	Req. No.	Defect Type	Description
1	2	RF 8	Omission	Missing a facility to allow the consulting of elements model, such as folders and hierarchical trees.
2			Omission	The requirements do not deal with defects treatments
3	3	RF 11/12	Ambiguity	It is not clear the difference between requirements 11 and 12
4	2	RF 5	Ambiguity	The terms participant and actor are being used to represent the same concept.
5			Omission	It is missing a specification for the user interface and the navigation mechanisms

Inspection Technique: scenario-based reading

Inspector receives a concrete set of instructions explaining how to read and what to look for in a software product.

- Increase the cost-effectiveness of inspections

More directed final result:

- Quality characteristics and reading approach defined *a priori*
- Technique induces productivity by reducing human influence on inspection results (i.e., ensure a more engineering approach)
- Provide models for writing documents of higher quality
- Easier to guarantee the inspector read the document in the correct way

Document coverage concerned with the reading technique

Cost/efficiency affected by the reading technique

Inspection Technique: scenario-based reading

More specifically, software reading is the individual analysis of a software artifact (e.g., requirements, design, code, test plans) to achieve the understanding needed for a particular task (e.g., defect detection, reuse, maintenance)

Scenario-based reading is:

document and notation specific

goal driven

tailorable to the project and environment

procedurally defined

focused to provide a particular document coverage

empirically verified to be effective for its use in inspections

Inspection Technique: scenario-based reading

Different Software Artifacts, Different Reading Techniques

perspective based reading (**PBR**):

for detecting defects in requirements documents

traceability based (horizontal/vertical) reading (**OORTS**):

for detecting defects in object oriented design in UML

usability based (heuristics) reading (**WDP**):

for detecting anomalies in user interface web screens

defect based reading (**DBR**):

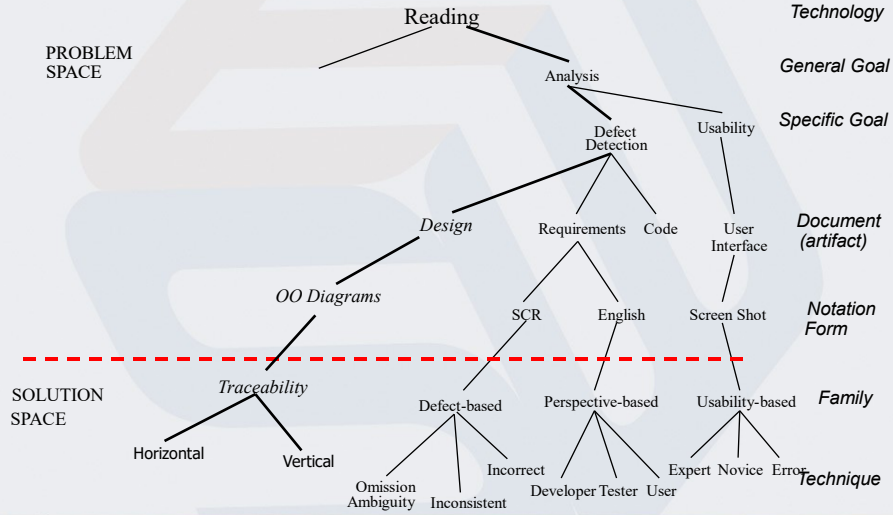
for detecting defects in requirements documents in SCR

scope based reading:

for constructing designs from OO frameworks

Reading techniques define an *approach* to be *tailored*.
There are different set of reading techniques.

Inspection Technique: scenario-based reading



Software Inspection Technique: summary

Technique / Features	*Ad-hoc*	Checklist based	Scenario-based reading
Notation	any	any	Language of "doing"
Systematic	no	partially	yes
Focused	no	no	yes
Controlled Improvement	does not allow	partially	yes
Adaptable	no	yes	yes
Training	no	partially	yes
Tailoring	no need	needed whether capturing specific quality characteristics	needed due to the used model
Introduction effort	low	medium	high
Document Coverage	no guarantee	depends on checklist and the inspector approach, but still hard to guarantee	Controlled by the technique
Cost-efficiency	depends on inspectors' experience	depends on inspectors' experience and checklist	depends on the technique, usually high

Travassos, G.H. (2014). Software Defects: Stay Away from them. Do Inspections!. QUATIC 2014. Keynote. (in press)

Perspective-Based Reading

- **What is PBR?**
  - Perspective-Based Reading (PBR) is an approach to inspection focusing on points of view.
  - The idea is that a document is correct if any potential stakeholder does not find a defect in it. Thus, a document should be read from various points of view (perspectives)
- **High Level Goals:**
  - make sure important perspectives are identified
  - make each reviewer responsible for a particular perspective
  - allow reviewers to build up expertise in different aspects of the document

Perspective-Based Reading

- **History:**
  - Piloted at NASA/Goddard Space Flight Center
  - Ongoing studies with students/professionals through universities: USC, UMCP, UMBC, Uni-Kl, NTNU, Uni-Lund, COPPE/UFRJ, USP-São Carlos, UFSCar, … …
  - Tutorial given at workshops and conferences: ICSE, NASA's Software Engineering Workshop, NASA's OSMA Software Assurance Symposium, …
  - Adapted for industrial use/training at Allianz Life Insurance, Bosch Telecom, Robert Bosch GmbH, CPqD, RioSoft, COPPE/UFRJ,...

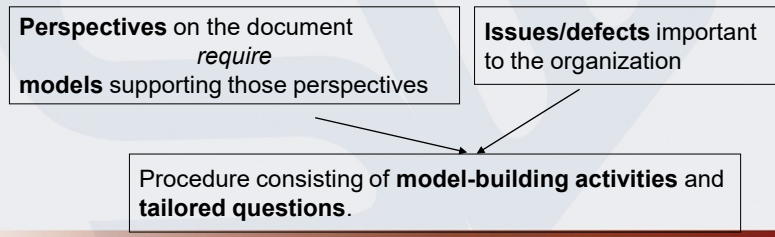
Perspective-Based Reading

- Under this approach, we create a set of scenarios such that:
  - Every PBR reviewer receives a **scenario** to guide his/her work.
    - a procedural description of activities and questions
    - that guides inspectors through the inspected document
  - Every scenario consists of two parts:
    - Construct a *model* of the document under review in order to increase understanding.
    - Answer *questions* about the model, focused on issues and problems of interest to the organization.

Defining PBR

To create the PBR scenarios:

1. Choose *perspectives* from which a document should be reviewed.
2. Choose a *model* of the document that makes sense for each perspective.
3. Choose the *defect types* of interest to the organization, and use that information to ask specific questions about each model.



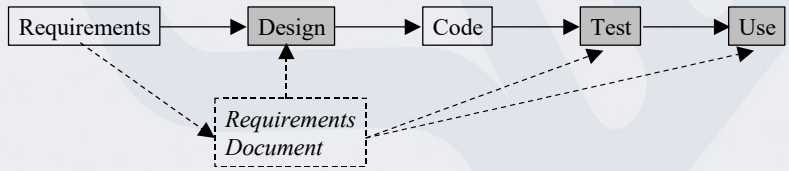
Defining PBR: Choosing perspectives

- PBR assigns different perspectives to different inspectors, i.e., each inspector of an inspection team reads a document from a particular perspective.
- Benefits:
  - focus the responsibilities of each inspector
  - Intends to minimize overlap among inspector responsibilities
  - Intends to maximize the union of defects found

Ad hoc coverage *PBR coverage*

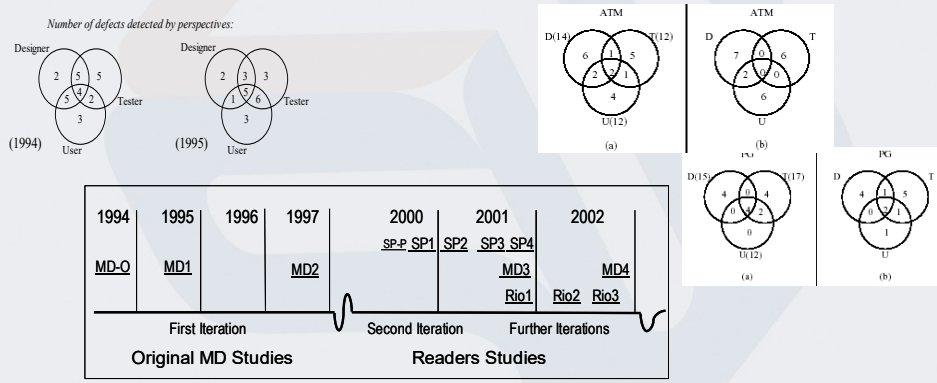
Defining PBR: Choosing perspectives

- An example of *perspectives*: Each user of a requirements document has his or her own needs for it:
  - a designer, who uses it to produce an OO model for the system;
  - a tester, who produces a test plan to ensure that the system meets the requirements;
  - a user, who has to make sure the requirements adequately capture the functionality needed in the final system;
  - …



Defining PBR: Choosing perspectives

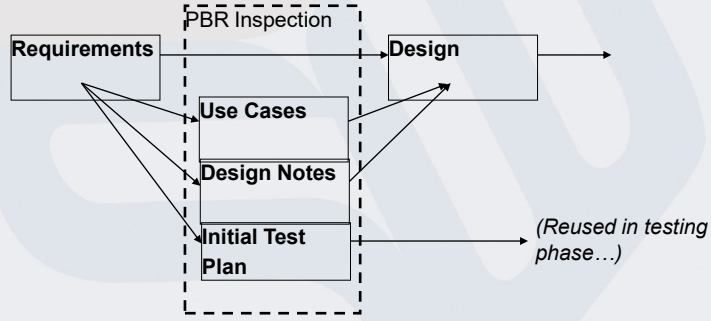
- PBR coverage data (examples from two studies):
  - Comes from a requirements document with 30 seeded defects…
  - Used in two experiments (1994 and 1995) with personnel from NASA Goddard Space Flight Center (replicated in 2001 and 2002 in Brazil)



Maldonado, J. C. ; Carver, J. ; Shull, F. ; Fabbri, S. C. P. F. ; Doria, E. S. ; Martimiano, L. ; Mendonça, M.; Basili, V. (2006). Perspective-Based Reading: A Replicated Experiment Focused on Individual Reviewer Effectiveness. Empirical Software Engineering , v. 11, n. 1, p. 119-142.

Defining PBR: Choosing models

- For each perspective, reviewers create a high-level model.
  - Have to emphasize that the purpose is defect detection.
- Choose models so that the extra effort spent on defect detection saves additional time in downstream phases.



Defining PBR: Choosing defect types

- Tailored specifically to requirements defects become…

(see Topic 1, slide 38, for original definitions),

Defect Classification	Description
Omission	(1) some significant requirement related to functionality, performance, design constraints, attributes or external interface is not included; (2) responses of the software to all realizable classes of input data in all realizable classes of situations is not defined; (3) missing sections of the requirements document; (4) missing labeling and referencing of figures, tables, and diagrams; (5) missing definition of terms and units of measures (ANSI, 1984).
Incorrect Fact	A requirement asserts a fact that cannot be true under the conditions specified for the system.
Inconsistency	Two or more requirements are in conflict with one another.
Ambiguity	A requirement has multiple interpretations due to multiple terms for the same characteristic, or multiple meanings of a term in a particular context.
Extraneous Information	Information is provided that is not needed or used.

Travassos, G. H., Shull, F. and Carver, J. Working with UML: A Software Design Process Based on Inspections for the Unified Modeling Language, in Advances in Computers, vol. 54, Academic Press, 2001

Defining PBR: Putting it all together

- List of defect types gets used to generate questions for each step of the model-building:

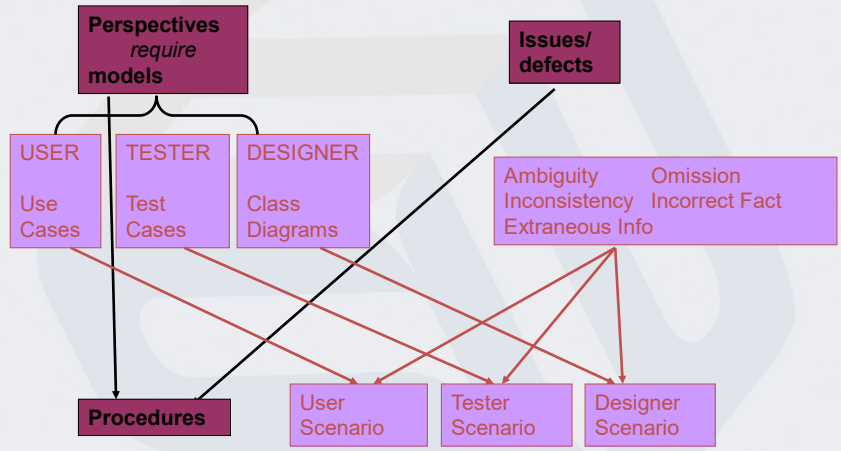
  - Step 1 for constructing model
    - Set of questions for this information
  - Step 2 for constructing model
    - Set of questions for this information
  - Step 3 ...

Perspective Based Reading (PBR)

- PBR explores the idea of different roles being played throughout the software development process;
- Therefore, PBR intends to provide a specific set of reading instructions accordingly an artifact "consumer" role (user, tester or designer)

- Looking for defects:
  - Will the requirements document attend the needs of its future "consumers"?
  - Will the information contained in the document allow the right system representation?
- If for some reason these questions can not be positively answered, a discrepancy should be identified and classified accordingly the type of defect.

Defining PBR techniques: example



Applying PBR: Changing the level of detail

- Level of detail of the techniques can be modified for:
  - Introducing PBR for the first time: Low level of detail used early to gather support
  - Tailoring for level of experience
    - Novices – more detail
    - Experts – only need the general perspective description
  - Using in different lifecycle models, such as spiral
    - Early iterations should have less detail, since the information is defined at a higher level of abstraction.

Applying PBR: Using the results

Individual review
- Individuals record defects
- Individuals record defects
- Individuals record issues

Team Meeting
- "Nominal meeting": lists are collated
- Team meets to forge consensus on final list
- Fagan-style meeting: issues serve as reminders while team undertakes defect detection together

Correction by Author

Applying PBR: Using the Data

- Metrics and observation based on use allow continual improvement able to:
  - add new perspectives of interest to the organization
  - change the mix of perspectives
    - if I expect a predominance of defects of a certain type, add more reviewers using relevant perspectives to capture these defects
  - change the underlying defect taxonomy
  - change the level of detail incorporated into the reading scenario

PBR "user" scenario

- User needs to make sure requirements adequately capture the functionality needed in the system
  - represent the users' view of the system (not an implementation view).
  - facilitate communication between system designers and the customers of a system.
  - help validate whether the system being described will work the way the customers expect it will.

PBR "user" scenario: High-level version

Rely on reader's experience with constructing use cases.

Define the set of functions that a user of the system should be able to perform.

Define the set of input objects necessary to perform each function, and the set of output objects that are generated by each function.

This may be viewed as writing down all operational scenarios or subsets of operational scenarios that the system should perform.

Start with the most obvious or nominal scenarios, and proceed to the least common scenarios or special/contingency conditions.

In doing so, ask yourself the following questions for each scenario:

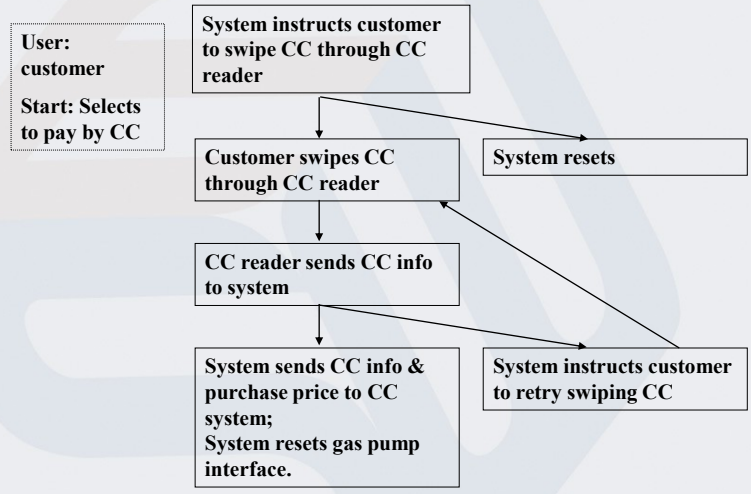
PBR “user” scenario: High-level version

1. Does the requirement/functional specification make sense from what you know about the application or from what is specified in the general description?
2. Are all the functions necessary to write the operational scenario described in the requirements or functional specifications?
3. Are the initial conditions for starting up the operational scenario clear and correct?
4. Are the interfaces between functions well defined and compatible (e.g., do the inputs of one function link to the outputs of the previous function)?
5. Might some portion of the operational scenario give different answers depending on how a requirement/functional specification is interpreted?
6. Can you get into a state of the system that must be avoided (e.g. for reasons of safety or security)?

PBR “user” scenario: Detailed version

- Choose a particular representation of functionality: use cases
  - is based on a **descriptive scenario** of how the user interacts with the system. It identifies events that can occur and describes the system response.
  - describes a particular set of system functionality by modeling the “dialogue” a user undertakes with the system.
  - is a complete and meaningful flow of events, taken from the perspective of a particular user of the system.
  - feeds into later stages of the lifecycle by helping to identify objects, develop test plans, and develop documentation
- The goal: Taken together, all use cases constitute all possible ways of using the system.

PBR "user" scenario: Detailed version



The "user" scenario: Detailed version

- **Use cases help in finding requirements defects**
  - Use cases provide a different representation of the same functionality described in natural-language requirements.
- Assumptions for defect detection:
  - If the natural-language requirements are well-specified, we should be able to convert them into use cases relatively easily.
  - If there are hidden problems with the natural-language requirements, they may prevent us from creating a clear, complete use case translation. These problems would probably also cause difficulties in constructing other artifacts based on the requirements, such as a design or code.
- The process of constructing use cases can be used as an aid in identifying defects.

PBR "user" scenario: Detailed version

- First major component: **participants**
  - entities in the environment (i.e. external to the system)
  - that interact with the system to achieve some functionality.
  - (often known as *actors*).
- Participants can be: human users, other systems, external organizations, external devices, etc., which interact with the system.
- Some participants initiate events; others interact with the system as a result of other events.
- When users are involved: a participant is a role (i.e. a characteristic way of interacting with the system) NOT a specific person.

PBR "user" scenario: Detailed version

- Second major component: **functionality**
- More than anything else, use cases are concerned with describing system behavior.
- Problem: Very hard to specify an exact definition for the intuitive way in which we usually describe system functionality.
- Lot of variation in how formally use case concepts are applied to describing a system: very loosely in some situations, very rigorously in others.
- We ask for functionality to be described at 2 levels:
  - product functions (What features does the product have? What are the specific activities it can do?)
  - use cases/user scenarios (What can customers use the system for?)

PBR "user" scenario: Detailed version

- **Step 1**
  - **General instructions:** Identify the participants in the requirements.
    - **Specific instructions:** (guidelines for recognizing participants)
  - **Questions:**
    - Q1.1 Are multiple terms used to describe the same participant in the requirements?
    - Q1.2 Is the description of how the system interacts with a participant inconsistent with the description of the participant? Are the requirements unclear or inconsistent about this interaction?
    - Q1.3 Have necessary participants been omitted? That is, does the system need to interact with another system, a piece of hardware, or a type of user that is not described?
    - Q1.4 Is an external system or a class of "users" described in the requirements which does not actually interact with the system?

PBR "user" scenario: Detailed version

- Which user groups use the system to perform a task?
- Which user groups are needed by the system in order to perform its functions?
- Which are the external systems that use the system in order to perform a task?
- Which are the external systems that are managed or otherwise used by the system in order to perform a task?
- Which are the external systems or user groups that send information to the system?
- Which are the external systems or user groups that receive information from the system?

Form A - Participants and use cases in new requirements

Reviewer Name:

Document Reviewed:

Participants	Involved in which use cases?	System Functionalities
1. **Credit card reader**		F1.
2. **Credit card system**		F2.
3. **Gas pump**		F3.
4. **Gas pump interface**		F4.
5. **Cashier interface**		F5.
6. **Customer**		F6.
7. **Cashier**		F7.
8.		F8.
9.		F9.
10.		F10.
11.		F11.
12.		F12.
13.		F13.
14.		F14.

PBR "user" scenario: Detailed version

- **Step 2.**
  - **General goal:** Describe system functionality.
    - **Specific goal:** (Procedure for identifying specific system product functions and understanding how they contribute to use cases)
  - **Questions:**
    - Q2.1 Are the start conditions for each use case specified at an appropriate level of detail?
    - Q2.2 Are the class(es) of users who use the functionality described, and are these classes correct and appropriate?
    - Q2.3 Is there any system functionality that should be included in a use case but is described in insufficient detail or omitted from the requirements?
    - Q2.4 Has the system been described sufficiently so that you understand what activities are required for the user to achieve the goal of a use case? Does this combination of activities make sense, based on the general description and your domain knowledge? Does the description allow more than one interpretation as to how the system achieves this goal?
    - Q2.5 Do the requirements omit use cases that you feel are necessary, according to your domain knowledge or the general description?

Form A - Participants and use cases in new requirements

Reviewer Name:

Document Reviewed:

Participants	Involved in which use cases?
1. Credit card reader	
2. Credit card system	
3. Gas pump	
4. Gas pump interface	
5. Cashier interface	
6. Customer	
7. Cashier	
8.	
9.	
10.	
11.	
12.	
13.	
14.	

System Functionalities
F1. Update inventory
F2. Bill at purchase
F3. Bill on monthly bill
F4. Accept credit card payment
F5. Accept cash payment
F6. Receive monthly bill
F7.
F8.
F9.
F10.
F11.
F12.
F13.
F14.

Form B - Use Case

Reviewer Name:

Document Reviewed:

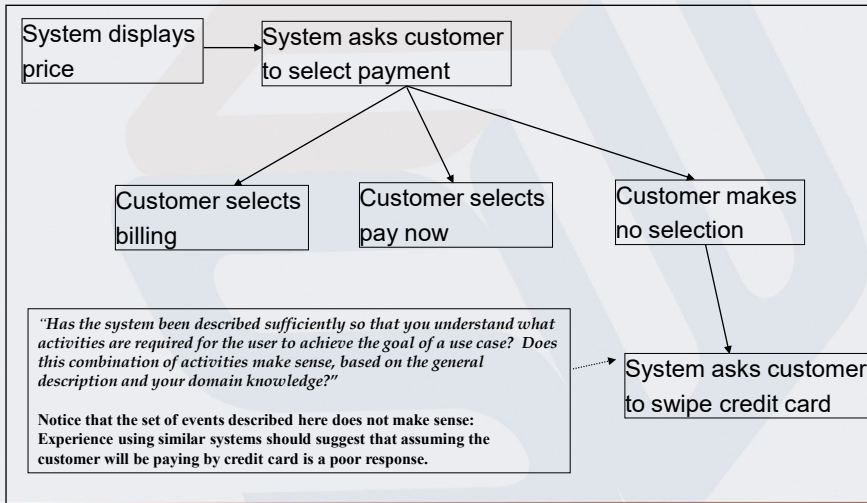
Use Case Name: Pay for gas at time of purchase

User: Customer

Start: Replace nozzle

Use Case Number: 1

Involves which functionalities? 2 | 3 | 4 | 5 | | |



"Has the system been described sufficiently so that you understand what activities are required for the user to achieve the goal of a use case? Does this combination of activities make sense, based on the general description and your domain knowledge?"

Notice that the set of events described here does not make sense: Experience using similar systems should suggest that assuming the customer will be paying by credit card is a poor response.

PBR "user" scenario: Detailed version

- **Step 3.**
  - **General goal:** Cross-index participants and functionality.
    - **Specific goal:** (Procedure for helping identify which participants are involved in which system functionalities)
  - **Questions:**
    - Q3.1 Is it clear from the requirements which participants are involved in which use cases?
    - Q3.2 Based on the general requirements and your knowledge of the domain, has each participant been connected to all of the relevant use cases?
    - Q3.3 Are participants involved in use cases that are incompatible with the participant's description?
    - Q3.4 Have necessary participants been omitted (e.g., are there use cases which require information that cannot be obtained from any source described in the requirements)?

Form A - Participants and use cases in new requirements

Reviewer Name:

Document Reviewed:

Participants	Involved in which use cases?	System Functionalities
1. Credit card reader	1	F1. Update inventory
2. Credit card system	1	F2. Bill at purchase
3. Gas pump		F3. Bill on monthly bill
4. Gas pump interface	1	F4. Accept credit card payment
5. Cashier interface	1	F5. Accept cash payment
6. Customer	1	F6. Receive monthly bill
7. Cashier	1	F7.
8.		F8.
9.		F9.
10.		F10.
11.		F11.
12.		F12.
13.		F13.
14.		F14.