Chapter - Five

UML in Analysis and Design

1

What UML to build under system Analysis Vs. Design?

2

Unified Modelling Language (UML)

- The Unified Modeling Language (UML) is a **graphical language** for OOAD that gives a standard way to write a software system's blueprint.
- It helps to visualize, specify, construct, and document the artifacts of an object-oriented system. It is used to depict the **structures and the relationships** in a complex system.

3

UML: Analysis

- What UML model to build under analysis. The following:
- **These are:**
  - Use case Diagram
  - Sequence diagram
  - CRC(Class Responsibility and Collaboration)
  - Class diagram
  - Activity Diagram

4

UML: Design

- What UML model to build under Design.
- **These are:**
  - Refined sequence Diagram
  - Collaboration Diagram
  - State chart diagram
  - Component Diagram
  - Deployment Diagram
  - Refined class diagram
  - Physical Data Modelling (PDM)
  - Graphical User Interface

5

UML - Modeling Types

- Any system can have two aspects, **static and dynamic**. So, a model is considered as complete when both the aspects are fully covered.
- **Structural Modeling/Diagrams**
  - Structural modeling captures the static features of a system. They consist of the following –
  - Classes diagrams
  - Objects diagrams
  - Deployment diagrams
  - Component diagram

6

Cont:

- Structural model represents the framework for the system and this framework is the place where all other components exist. Hence, the class diagram, component diagram and deployment diagrams are part of structural modeling. They all represent the elements and the mechanism to assemble them.
- The structural model never describes the dynamic behavior of the system. Class diagram is the most widely used structural diagram.
- The structural diagrams represent the static aspect of the system. These static aspects represent those parts of a diagram, which forms the main structure and are therefore stable.

7

Class Diagram

- Class diagrams are the most common diagrams used in UML. Class diagram consists of classes, interfaces, relationships, and collaboration.
- Class diagrams basically represent the object-oriented view of a system, which is static in nature.
- A class is a set of objects that share a common structure and common behavior (the same attributes, operations and semantics).
- A class is an abstraction of real-world items.

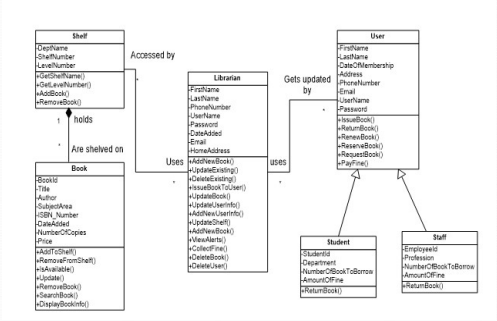
8

Cont....

- Classes are depicted as **boxes with three sections**:
  - the top one indicates **the name of the class**,
  - the middle one lists the **attributes of the class**, and
  - the third one lists the **methods.**
- Or a class can **have two sections**,
  - one for the **name** and
  - one for the **responsibilities**.

9

Analysis Class Diagram

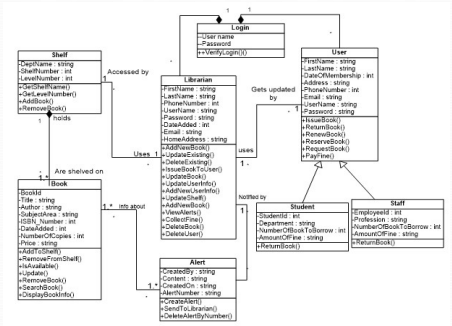


10

- To create and evolve a class model, you need to model these items:
  - Classes;
  - Responsibilities(Attributes and methods);
  - Associations;
  - Dependencies;
  - Inheritance relationships;
  - Composition associations; and
  - Association classes.

11

Design Class Diagram



12

➢ Object diagram

- Object diagrams can be described as an instance of class diagram. Thus, these diagrams are more close to real-life scenarios where we implement a system.
- Object diagrams are a set of objects and their relationship is just like class diagrams. They also represent the static view of the system.
- The usage of object diagrams is similar to class diagrams but they are used to build prototype of a system from a practical perspective.

13

➢ Component diagram

- Component diagrams represent a set of components and their relationships.
- Used to model physical aspects of a system
- These components consist of classes, interfaces, or collaborations.
- Component diagrams represent the implementation view of a system.
- During the design phase, software artifacts (classes, interfaces, etc.) of a system are arranged in different groups depending upon their relationship. Now, these groups are known as components.
- Finally, it can be said component diagrams are used to visualize the implementation.

14

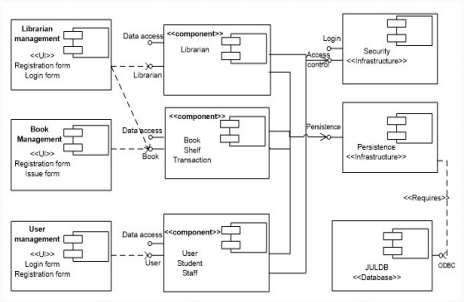
➢ Cont…

- These diagrams also show the externally visible behavior of the component by displaying the interfaces of the components.
- Components
- Interfaces
- Dependency relationship.

- Component diagrams
  - Models business and technical software architecture
  - Uses components defined in the composite structure diagrams, in particular their ports and interfaces

15

Component diagram for Library Management system



16

Deployment Diagrams

- Deployment diagrams are a set of nodes and their relationships.
- These nodes are physical entities where the components are deployed.

Used to visualize the topology of the physical components of a system

where the software components are deployed.

- A deployment diagram shows processors, devices and connections. Each model contains a single deployment diagram which shows the connections between its processors and devices and its processes to processors.

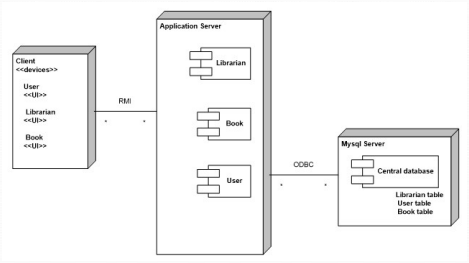
17

Cont…

- Deployment diagrams
  - Models the physical software architecture, including issues such as the hardware, the software installed on it and the middleware
  - Gives a static view of the run-time configuration of processing nodes and the components that run on those nodes

18

Deployment diagram for Library Management system



19

➢ Behavioral Modeling

- Behavioral diagrams basically capture the dynamic aspect of a system.
- It represents the interaction among the structural diagrams.
- Dynamic aspect can be further described as the changing/moving part of the system
- All the above show the dynamic sequence of flow in a system.
  - Use case diagrams
  - Sequence Diagram
  - Collaboration Diagram
  - State chart Diagram
  - Activity diagrams

20

Use Case Diagram

- Use case diagrams are a set of use cases, actors, and their relationships. They represent the use case view of a system.
- **Use case diagram designed under analysis**

21

Cont…

- **For use case we analysis:**
  - Use case ID: example UC1: Login
  - Pre-condition
  - Post-condition
  - Main course of action
  - Alternative course of action
  - Include and Extend

22

Include and Extend

- The relationship between Use cases can be reused and extended in two different fashions: **extends and include.**
- In the cases of **"include"** relationship, we define that one use case invokes the steps defined in another use case during the course of its own execution. Hence this defines a relationship that is similar to a relationship between two functions where one makes a call to the other function.
- The **"extends"** relationship is kind of a generalization-specialization relationship. In this case a special instance of an already existing use case is created. The new use case inherits all the properties of the existing use case, including its actors.

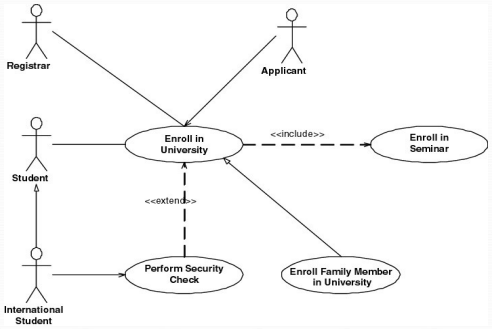
23

Cont…

- **Extend :<<extend>>**
  - When the extending use case activity sequence is completed, the base use case continues.
  - An extending use case is, effectively, an alternate course of the base use case.
  - Ex. *Enroll in University* is the base use case and *Perform Security Check* is the extending use case. Step 7
- **Include: <<include>>**
  - formerly known as a uses relationship in UML v1.2 and earlier
  - is a generalization relationship denoting the inclusion of the behavior described by another use case
  - is the invocation of a use case by another one
  - Used whenever one use case needs the behavior of another
  - Ex. If a student is enrolled in a university, he/she can enroll in a seminar . Step 11

24

Cont…



25

Cont…

- **Inheritance between use cases**
  - Use cases can inherit from other use case. It is not as common as the use of either extend or include associations, but it is still possible
  - The inheriting use case would completely replace one or more of the courses of action of the inherited use case
  - **Ex. that *Enroll Family Member in University* inherits from the *Enroll in University* use case** (in this case, to reflect that new business rules are applied when the family member of a professor is enrolling at the university)
  - It should have a name, description, and identifier, and it should indicate from which use case it inherits (If something is not replaced, then leave that section blank, assuming it is inherited from the parent use case)
- **Inheritance between actors**
  - an actor on a use case diagram can inherit from another actor
  - **Ex. *International Student* actor inherits from *Student*.**
  - International student is subjected to different rules and policies (pay more and undergo a security check during registration process)

26

Example:

- **Login**
- ID: UC1
- Actor: librarian
- Pre-condition: the librarian/user should have their user name and password to login.
- Post-condition: the main page of the system should display on the screen.
- Include: user database.

27

Example: cont…

➢ **Main course of action: step by step**

1. An authorized librarian/user want to login to the system.
2. The librarian/user click on login form.
3. The system asks user name and password to the login to the system.
4. Then enter his/her username and password to login form and click on login.
5. The system checks and authenticates the librarian/user.
6. The database accepts the username and password of the librarian/user.
7. Then the main page will be displayed to the users.

28

Example: cont…

➢ **Best practices in writing Alternate Courses of Action**

- **Briefly describe the condition:** description of the condition that must be met to invoke the alternate course
- **Indicate that the alternate course exists within the basic course of action. *(Specify the alternate course name)***
- **Set a consistent step numbering strategy*:*** each step starts with the letter of the ***alternate course***, followed by the number of the step in the ***basic course of the use case*** it replaces
- **You need an identification scheme:** identify the first alternate course as *A*, the second as *B*, and so on. Also, notice the numbering scheme for the steps of the alternate course.
- **End the alternate course:** indicate either that the use case ends or the use case continues at another step

29

Example: cont…

➢ **Alternative course of action**

A.5 If the user type wrong password or username.

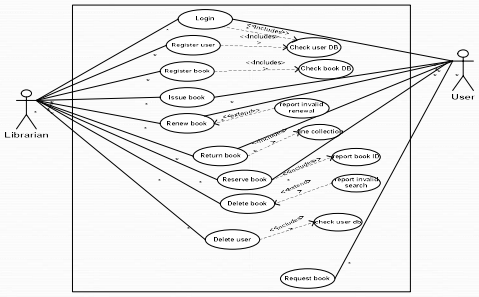
A.6.The login controller initiates login.

A.7.The database authenticate the librarian/user.

A.8.The system display login form to the librarian/user continuously.

30

Software
Use case diagram



Interaction diagrams is part of Behavioral diagram

- Interaction is basically a message exchange between two UML components. The following diagram represents different notations used in an interaction.
- Interactions can be of two types –
  - **Sequential** (Represented by sequence diagram)
  - **Collaborative** (Represented by collaboration diagram)

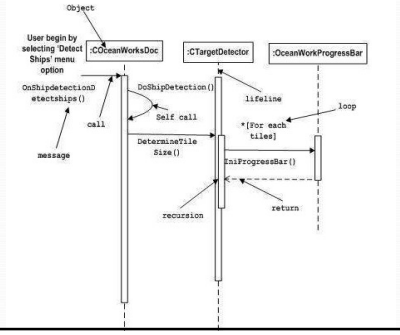
Cont…

- ***Sequence Diagrams:*** A sequence diagram is a graphical view of a scenario that shows object interaction in **a time based sequence**, what happens first, what happens next.
- Sequence diagrams establish the roles of objects and help provide essential information to determine class responsibilities and interfaces.
- A sequence diagrams in design phases are a refined diagrams with additional details of activities.

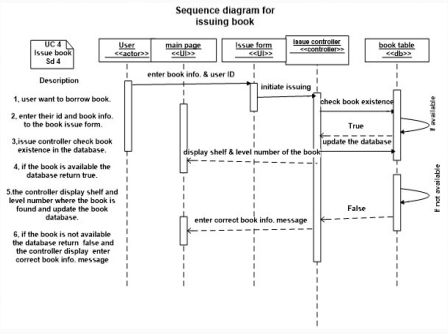
Rules in drawing a sequence diagram

- the **horizontal arrows** shows the **message** : the first message starts in the top left corner; the next message appears just below that one
- The **boxes across the top of the diagram** represent **classifiers** or their instances; typically use cases, objects, classes, or actors
  - **Objects have labels** in the standard UML format ***name: ClassName*** (name is optional, if object has no name "Anonymous") and ***object labels are underlined***
  - **Classes have labels** in the format ***ClassName***, and
  - **Actors have names** in the format ***Actor Name***.
- The **dashed lines hanging from the boxes** are called **object lifelines**, representing the life span of the object during the scenario being modeled.
- **The long, thin boxes on the lifelines** are **activation boxes**, also called **method-invocation boxes**, which indicate processing is being performed by the target object/class to fulfill a message.

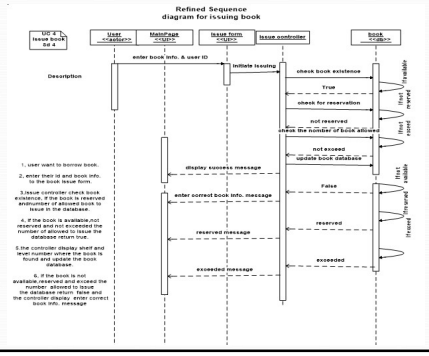
Cont…



Analysis Sequence Diagram



Design Sequence Diagram



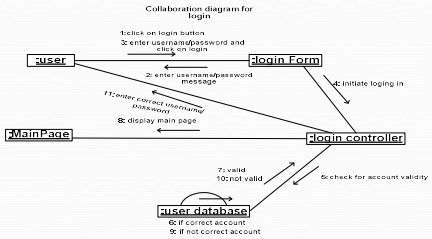
37

Collaboration Diagram

- It represents the structural organization of a system and the messages sent/received. Structural organization consists of objects, links and messages.
- Order of messages that implement an operation or a transaction. They can also contain simple class instances and class utility instances
- The purpose of collaboration diagram is similar to sequence diagram. However, the specific purpose of collaboration diagram is to visualize the organization of objects and their interaction.

38

Cont…



39

State Machine Notation

- State machine describes the different states of a component in its life cycle. The notations are described in the following diagram.
- State machine is used to describe different states of a system component.
- The state can be active, idle, or any other depending upon the situation.

40

Cont….

- *State_chart_diagram*: state chart diagrams model the dynamic behavior of individual classes or any other kind of object.
- They show the sequence of states that an object goes through the events that cause a transition from one state to another and the actions that result from a state change.
- A state chart diagram is typically used to model the discrete stages of an objects lifetime.

41

Essential elements of state chart diagram

- *State:* - A state represents a condition or situation during the life of an object during which it satisfies some condition or waits for an event. Each state represents a cumulative history of its behavior.
- States can be shared between state machines. Transitions cannot be shared

42

cont....

- *Start state:-* A start state (also called an "initial state") explicitly shows the beginning of the execution of the state machine on the state chart diagram or beginning of the workflow on an activity diagram.
- Normally, one outgoing transition can be placed from the start state.
- *End state:* - An end state represents a final or terminal state on an activity or state chart diagram.
- Transitions can only occur into an end state.
- The end state icon is a filled circle inside a slightly larger unfilled circle that may contain the name (End process).

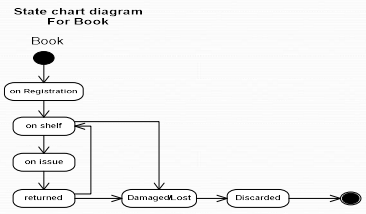
43

cont....

- *State transition:-* A state transition indicates that an action in the source state will perform certain specified actions and enter the destination state when a specified event occurs or when certain conditions are satisfied.

44

cont....



45

Activity diagram

- Activity diagram describes the flow of control in a system. It consists of activities and links. The flow can be sequential, concurrent, or branched.
- Activities are nothing but the functions of a system. Numbers of activity diagrams are prepared to capture the entire flow in a system.
- Activity diagrams are used to visualize the flow of controls in a system. This is prepared to have an idea of how the system will work when executed.

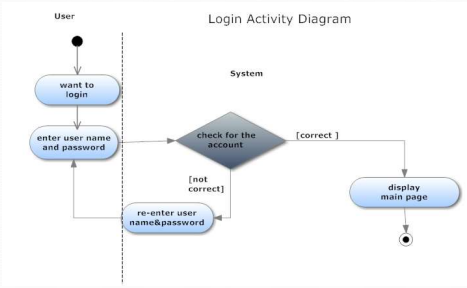
46

Cont...

- **Basic notations of Activity Diagram**
  - **Initial node:** filled circle is the starting point of the diagram
  - **final node.** The filled circle with a border is the ending point (0 , >0)
  - **Activity.** The rounded rectangles represent activities that occur (be physical, such as *Inspect Forms*, or electronic, such as *Display Create Student Screen)*
  - **Flow/edge.** The arrows on the diagram
  - **Condition.** Text such as [*Incorrect Form*] on a flow
  - **Decision.** A diamond with one flow entering and several leaving

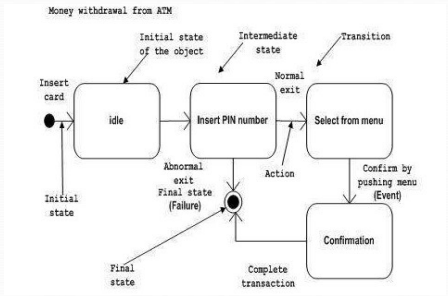
47

Cont....



48

Cont…



49

Physical Data Modelling (PDM)

- Purpose of PDM diagrams:
  - design the internal schema of a database,
  - depicting the data tables,
  - the data columns of those tables, and
  - the relationships between the tables. (implemented using keys)

50

- **In designing a physical data model:**
  - **Identify tables:** Tables are the database equivalent of classes
  - **Normalize tables: to** ensure that data are stored in only one place/table
  - **Identify columns:** A column is the database equivalent of an attribute.
  - **Identify stored procedures:** A stored procedure is conceptually similar to a global method implemented by the database.
  - **Identify relationships.** There are relationships between tables just like there are relationships between classes.
  - **Assign keys.** A key is one or more data attributes that uniquely identify a row in a table. <<PK>> Primary key and foreign keys <<FK>>.
  - **Finally show all the tables with their fields/attributes description**

51

dbo.Student: Tab...P_DATA\ORSC.MDF)*

Column Name	Data Type	Allow Nulls
No	bigint	
FirstName	varchar(50)	
LastName	varchar(50)	
Sex	char(6)	
BirthDate	varchar(50)	
University	varchar(50)	
Department	varchar(50)	
Year	int	
Address	ntext	
PhoneNumber	varchar(50)	
Mobile	varchar(50)	
Email	varchar(50)	
IDNumber	varchar(50)	
Password	varchar(50)	
Photo	image	☑
JoinDate	datetime	

52

Graphical User Interface

- help the user to navigate through the system and enter data to the database, access and retrieve data from the database.
- Design class diagram will be converted to tables and tables will be converted to GUI
- Every table that is designed in the physical data model should have corresponding graphical user interface.
- **Properly designed GUI have the following benefits:**
  - easier to use, easier to train people to use it(reduce training costs)
  - less help people will need to use it (reduce your support costs)
  - users will like to use it (increase their satisfaction)

53

User Interface Design Tips and Techniques

- **Consistency:** make sure that your user interface works consistently
- **Set standards and stick to them**
- **Explain the rules**. users need to know how to work with the application
- **Navigation between screens is important**
- **Word your messages and labels appropriately:** primary source of information
- **Understand your widgets:** use the right widget for the right task
- **Look at other applications with a grain of salt:** make sure the standards you used are also used by other organizations
- **Use color appropriately**
- **Follow the contrast rule:** ensure readability
- **Use fonts.** Which are easy to read
- **Gray things out, do not remove them:** restrict access

54

- **Alignment of fields:** organize the fields in a way that is both visually appealing and efficient
- **Justify data appropriately:** right justify integers, decimal align floating point numbers, and left justifies strings
- **Do not create busy screens:** do not overcrowd
- **Group things on the screen effectively:** group them logically

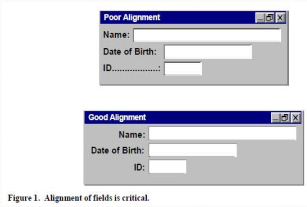


Figure 1. Alignment of fields is critical.

55

Relationships and Multiplicity

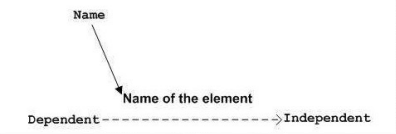
- A model is not complete unless the relationships between elements are described properly. The *Relationship* gives a proper meaning to a UML model. Following are the different types of relationships available in UML

- Dependency
- Association
- Generalization
- Aggregation
- Composition
- Extensibility

56

Dependency Notation

- Dependency is represented by a dotted arrow as shown in the following figure. The arrow head represents the independent element and the other end represents the dependent element.



57

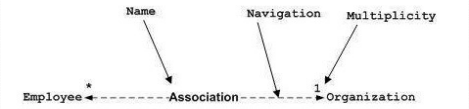
Example:

- The model elements at the two ends of the **dependency** are called client and supplier. The client is dependent on the supplier, such that a change in the supplier may result in a change in the client.
- A **dependency** between packages is used when the content of one package is dependent on the content of another package.

58

Association Notation

- Association describes how the elements in a UML diagram are associated. In simple words, it describes how many elements are taking part in an interaction.
- Association is represented by a dotted line with (without) arrows on both sides. The two ends represent two associated elements as shown in the following figure. The multiplicity is also mentioned at the ends (1, *, etc.) to show how many objects are associated.
- Association is used to represent the relationship between two elements of a system.



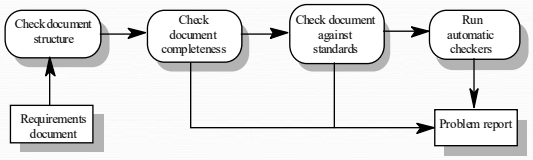
59

Generalizaton Notation

- Generalization describes the inheritance relationship of the object-oriented world. It is a parent and child relationship.
- Generalization is represented by an arrow with a hollow arrow head as shown in the following figure. One end represents the parent element and the other end represents the child element.

60

Pre-review checking



61

Review team membership

- Reviews should involve a number of stakeholders drawn from different backgrounds
  - People from different backgrounds bring different skills and knowledge to the review
  - Stakeholders feel involved in the RE process and develop an understanding of the needs of other stakeholders
- Review team should always involve at least a domain expert and an end-user

62

Review checklists

- Understandability
  - Can readers of the document understand what the requirements mean?
- Redundancy
  - Is information unnecessarily repeated in the requirements document?
- Completeness
  - Does the checker know of any missing requirements or is there any information missing from individual requirement descriptions?
- Ambiguity
  - Are the requirements expressed using terms which are clearly defined? Could readers from different backgrounds make different interpretations of the requirements?

63

Review checklists

- Consistency
  - Do the descriptions of different requirements include contradictions? Are there contradictions between individual requirements and overall system requirements?
- Organisation
  - Is the document structured in a sensible way? Are the descriptions of requirements organised so that related requirements are grouped?
- Conformance to standards
  - Does the requirements document and individual requirements conform to defined standards? Are departures from the standards, justified?
- Traceability
  - Are requirements unambiguously identified, include links to related requirements and to the reasons why these requirements have been included?

64

Checklist questions

- Is each requirement uniquely identified?
- Are specialised terms defined in the glossary
- Does a requirement stand on its own or do you have to examine other requirements to understand what it means?
- Do individual requirements use the terms consistently
- Is the same service requested in different requirements? Are there any contradictions in these requests?
- If a requirement makes reference to some other facilities, are these described elsewhere in the document?
- Are related requirements grouped together? If not, do they refer to each other?

65

Requirements problem example

- "4. EDDIS will be configurable so that it will comply with the requirements of all UK and (where relevant) international copyright legislation. Minimally, this means that EDDIS must provide a form for the user to sign the Copyright Declaration statement. It also means that EDDIS must keep track of Copyright Declaration statements which have been signed/not-signed. Under no circumstances must an order be sent to the supplier if the copyright statement has not been signed."

66

Problems

- Incompleteness
  - What international copyright legislation is relevant?
  - What happens if the copyright declaration is not signed?
  - If a signature is a digital signature, how is it assigned?
- Ambiguity
  - What does signing an electronic form mean? Is this a physical signature or a digital signature?
- Standards
  - More than 1 requirement. Maintenance of copyright is one requirement; issue of documents is another

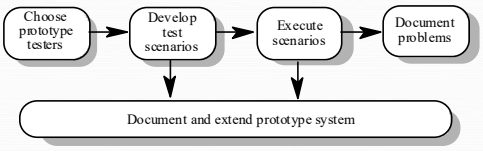
67

Prototyping

- Prototypes for requirements validation demonstrate the requirements and help stakeholders discover problems
- Validation prototypes should be complete, reasonably efficient and robust. It should be possible to use them in the same way as the required system
- User documentation and training should be provided

68

Prototyping for validation



69

Prototyping activities

- **Choose prototype testers**
  - The best testers are **users** who are fairly experienced and who are open-minded about the use of new systems. End-users who do different jobs should be involved so that different areas of system functionality will be covered.
- **Develop test scenarios**
  - **Careful planning** is required to draw up a set of test scenarios which provide broad coverage of the requirements. End-users shouldn't just play around with the system as this may never exercise critical system features.
- **Execute scenarios**
  - The users of the **system work**, usually on their own, to try the system by executing the planned scenarios.
- **Document problems**
  - Its usually best to define some kind of **electronic or paper problem report** form which users fill in when they encounter a problem.

70

User manual development

- Writing a user manual from the requirements forces a **detailed requirements analysis** and thus can reveal problems with the document
- **Information in the user manual**
  - Description of the functionality and how it is implemented
  - Which parts of the system have not been implemented
  - How to get out of trouble
  - How to install and get started with the system

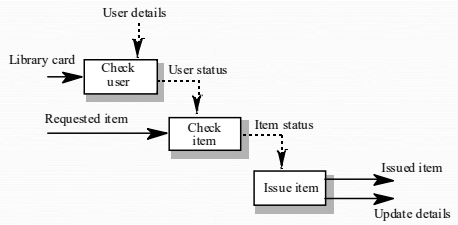
71

Model validation

- Validation of system models is an **essential part** of the validation process
- **Objectives of model validation**
  - To demonstrate that **each model** is **self-consistent**
  - If there are several models of the system, to demonstrate that these are **internally and externally consistent**
  - To demonstrate that the models accurately **reflect** the **real requirements** of system stakeholders
- Some checking is possible with automated tools
- Paraphrasing the model is an effective checking technique

72

Data-flow diagram for Issue



73

Paraphrased description

Check user	
Inputs and sources	User's library card from end-user
Transformation function	Checks that the user is a valid library user
Transformation outputs	The user's status
Control information	User details from the database
Check item	
Inputs and sources	The user's status from Check user
Transformation function	Checks if an item is available for issue
Transformation outputs	The item's status
Control information	The availability of the item
Issue item	
Inputs and sources	*None*
Transformation function	Issues an item to the library user. Items are stamped with a return date.
Transformation outputs	The item issued to the end user Database update details
Control information	Item status - items only issued if available

74

Requirements testing

- Each requirement should be testable i.e. it should be possible to define tests to check whether or not that requirement has been met.
- Inventing requirements tests is an effective validation technique as missing or ambiguous information in the requirements description may make it difficult to formulate tests
- Each functional requirement should have an associated test

75

Test case definition

- What usage scenario might be used to check the requirement?
- Does the requirement, on its own, include enough information to allow a test to be defined?
- Is it possible to test the requirement using a single test or are multiple test cases required?
- Could the requirement be re-stated to make the test cases more obvious?

76

Test record form

- **The requirement's identifier**
  - There should be at least one for each requirement.
- **Related requirements**
  - These should be referenced as the test may also be relevant to these requirements.
- **Test description**
  - A brief description of the test and why this is an objective requirements test. This should include system inputs and corresponding outputs.
- **Requirements problems**
  - A description of problems which made test definition difficult or impossible.
- **Comments and recommendations**
  - These are advice on how to solve requirements problems which have been discovered.

77

Requirements test form

Requirements tested: 10.(iv)

Related requirements: 10.(i), 10.(ii), 10.(iii), 10.(vi), 10. (vii)

Test applied: For each class of user, prepare a login script and identify the services expected for that class of user.

The results of the login should be a web page with a menu of available services.

Requirements problems: We don't know the different classes of EDDIS user and the services which are available to each user class. Apart from the administrator, are all other EDDIS users in the same class?

Recommendations: Explicitly list all user classes and the services which they can access.

78

Hard-to-test requirements

- **System requirements**
  - Requirements which apply to the system as a whole. In general, these are the **most difficult requirements** to validate irrespective of the method used as they may be influenced by any of the functional requirements. Tests, which are not executed, cannot test for non-functional system-wide characteristics such as usability.
- **Exclusive requirements**
  - These are requirements which exclude specific behaviour. For example, a requirement may state that system failures must never corrupt the system database. It is not possible to test such a requirement exhaustively.
- **Some non-functional requirements**
  - Some non-functional requirements, such as reliability requirements, can only be tested with a large test set. Designing this test set does not help with requirements validation.

79

Key points

- Requirements validation should focus on checking the final draft of the requirements document for conflicts, omissions and deviations from standards.
- Inputs to the validation process are the requirements document, organisational standards and implicit organisational knowledge. The outputs are a list of requirements problems and agreed actions to address these problems.
- Reviews involve a group of people making a detailed analysis of the requirements.
- Review costs can be reduced by checking the requirements before the review for deviations from organisational standards. These may result from more serious requirements problems.

80

Cont…

- Checklists of what to look for may be used to drive a requirements review process.
- Prototyping is effective for requirements validation if a prototype has been developed during the requirements elicitation stage.
- Systems models may be validated by paraphrasing them. This means that they are systematically translated into a natural language description.
- Designing tests for requirements can reveal problems with the requirements. If the requirement is unclear, it may be impossible to define a test for it.

81