FIT2001- System development- Note

Systems Development (Monash University)

Seminar 7: Investigating system requirements – Prototyping Usability of systems

Prototyping:

- Quickly mocking up the future system functionality
- Uses visuals to describe how a system should behave and look.
- Can be experimental or evolutionary
- Can be horizontal or vertical
- Advantage:
    - Explore ideas before you invest in them (improve communication, reduce risk)
    - Saves time and money
    - Proof of concept
    - Design & Technical exploration
- Disadvantage:
    - Make the users think the system is developed (manage expectations)
    - Create a system that doesn't scale
    - Waste time (as developers spend a great deal of time making throw away prototypes look good)

Scoping a Prototype:

- Include complex interactions, new functionality, changes in workflow
- The functionality that will be used most of the time
- Find the story: user stories, different persona experiences
- Plan your iterations: Start broadly, then drill down for some functionality
- Choose the appropriate fidelity
    - Visual (style)
    - Functional (Interactions)
    - Content (real)

The Prototyping spectrum

- Low fidelity
    - create rough paper based mock-ups
    - gets feedback on design approaches and concepts
    - lets you make changes easily and quickly
- Medium fidelity
    - Increased fidelity with computer based tools
    - demonstrates behaviour of the application
    - simulates interactions
- High fidelity
    - Most realistic, often mistaken for final product
    - Excellent reference for developers
    - Great for usability testing and training
    - Learning curve for developers

Usability :

- Systems that are User-Friendly & Easily Utilised→Degree to which something is easy to use & good fit for its users:
    - Effectiveness: accuracy and completeness
    - Efficiency: resources expended in relation to the 'effectiveness'
    - Satisfaction: the comfort and acceptability of the work system to its users
- Evaluating Usability – 5 criteria:
    - **Learnability**: How easy is it for users to accomplish basic tasks at the first time?
    - **Efficiency**: Once users have learned the design, how quickly can they perform tasks?
    - **Memorability**: When users return to the design after a period of not using it, how easily can they re-establish proficiency?
    - **Errors**: How many errors do users make? How severe are these errors? How easily can they recover from the errors?
    - **Satisfaction**: How pleasant is it to use the design?
- Avoid:
    - Expensive
    - Time-Consuming
    - Creativity Killer
    - Focus Groups
    - Customer Satisfaction Surveys
- Usability Rationale:
    - Improve User Efficiency
    - Feel more in Control (Actors)
    - Improve User Satisfaction
    - Improve Sales of Certain Products
    - Improve Actual Usage of Systems
- Ways to evaluate:
    - Formative evaluation
        - Users experience prototypes and identify usability problems
        - Evaluation by HCI experts
    - Summative evaluation
        - Experts observe → Takes place post development
        - Quantitative results collected
- Usability Measurements:
    - Time to Learn
    - Speed of Performance
    - Rate of Errors
    - Retention Overtime
    - Subjective Satisfaction

Seminar 8: Designing the User Interface

User interface (UI):

- Definition
  - UI represents the ONLY part of a software system that end-users only:
    - – Visualise
    - –And interact with
- Such interactions involve INPUTS and OUTPUTS from end-users:
  - purpose (input, dialog box, report)
  - user characteristics (users with disability, novice/experienced)
  - device (e.g. mobile phone screen size)
- Impacts:
  - Impact 1:Usability
    - can be drastically improved through good USER INTERFACE (UI)
    - easy to learn, navigate (use), and easy to understand
  - Impact 2: User Acceptance
  - A satisfactory UI is essential to promote acceptance of a system

User-centred design approach:

- Early and continuous focus on users and their work (Use personas)
- Evaluate all designs to ensure usability
- Design iteratively

Personas:

- Explore the psychology of an imagined user's interaction with the product
- Creating products for specific NOT generic users, provides a clear vision rather than unfocussed goal

How do you create Personas?

1. Collect information about your users: understand the target audience's mindsets, motivations, and behaviours(interviews, workshops, questionnaires)
2. Identify behavioural patterns from research data – find patterns in data to help group users
3. Create personas and prioritise them
   a. assemble personas around patterns –add just enough detail to characterise the user base
   b. if you have multiple personas define the primary perso
4. Identify relevant scenarios for the personas
   a. by pairing the personas with the scenarios, you can gather requirements and design relevant solutions
5. Share your findings and socialise personas among stakeholders
   a. see the value in them, should be front and centre of the design process

Why are Personas important?

- Build Empathy

    - Helps users seem more real - designers empathise and build for their users
- Provide Direction For Making Design Decisions
    - Helps focus design decision on users – don't build it for yourself or a generic user
- Communicate Research Findings
    - Team on the same page, communicates information in an easy to understand format

Ben Shneiderman 8 Golden Rules of interface design:

1. Strive for Consistency→Arrangement, Names & Shapes, Sequence
2. Cater for Diverse Users → Self-Explanatory
3. Offer Informative Feedback→Show that user's actions were received (Informative & Clear & Concise).
4. Design Dialogues that need Closure→Organising the sequences of actions to ensure that users know when a conversation/task is at end.
5. Prevent Errors→Inform users of errors, including reasons & a possible solution.
6. Permit Easy Reversal of Actions→Users capable to cancel/reverse an action.
7. Reduce Short-Term Memory Load→Prevent users from over- memorising information.
8. Support Internal Locus of Control→Experienced users needs to be in charge during interactions.

Jakob Nielsen→10 Usability Heuristics for interface design

1. Visibility of system status
2. Match between system & reality
3. User control & freedom
4. Consistency & standards
5. Error prevention
6. Recognition > Recall
7. Flexibility & Efficiency
8. Aesthetic & minimalist design
9. Help user recover from errors
10. Help & documentation

Design Principles:

- Minimise the pain
- Illuminate a path to completion
- Consider the context
- Consistent Communication

Seminar 9: Use case realisation – Sequence diagrams

Object Oriented (OO) Design

- Bridge from requirements to solution→Focuses on how to build & what structural components.
- A process by which detailed OO Design models (blueprints) are created – extends the requirements models
- The design models are used to build the new system, develop the database, user-interfaces, networks, controls and security

Three-layer client-server architecture:

- View (Client) layer
  - requests the resources, equipped with a user interface (usually a web browser) for presentation purposes
  - HTML5, JavaScript, CSS
- Domain layer (Application logic)
  - task it is to provide the requested resources, but by calling on another server
  - is usually hosted on one or more application servers, but can also be hosted in the cloud
  - Java, .NET, C#, Python, C++, etc.
- Data layer
  - provides the application server with the data that it requires
  - MySQL, Oracle, PostgreSQL, Microsoft SQL Server, MongoDB

Design Class

- Extends domain class
- involve an Artificial Class which is not needed by analysts and hence not included in the Domain Class model
- Contains THREE parts (i.e. Class name, attributes, methods)

Design class and Objects

- A DESIGN CLASS can have Multiple Objects that are created RUNTIME
- A DESIGN CLASS is a collection of the same type of OBJECTS, with:
  - the same properties & the same methods
- An OBJECT is an instance of a Design Class during RUN TIME
- An OBJECT has its:
  - own identity, attribute values
  - perform operations (methods) defined by the class
- An object can communicate to another object by passing messages
- An object knows:
  - its attributes
  - the actions (methods) can perform
  - with which other objects it is associated
  - hence represents a SELF-CONTAINED UNIT

Class	Object
Class represents a THING of interest to users and designers for which info is collected	Object represents an instance of a class
Class is the template based on which Objects are created	Objects give a LIFE to a CLASS
It is conceptual and exists in specifications. Hence, it does not occupy a MEMORY LOCATION	It occupies a MEMORY LOCATION as it is created during program run time
It cannot be manipulated because it is not available in MEMORY	It can be manipulated

Attributes of Design Class:

- Class Name
  - The name of the class appears in the first partition.
- Class Attributes
  - Describe properties (characteristics)
  - Attributes are shown in the second partition.
  - The attribute type is shown after the colon.
  - Attributes map onto member variables (data members) in code.
  - Objects can inherit attributes from other objects. Examples: -startingJobCode: integer = 01
  - Lower case camelBack notation
  - May have Property like {key}
  - Class level attribute (GLOBAL VARIABLES):
    - is associated with a class, rather than with an object.
    - All objects of a class share that attribute. It is COMMON to all objects of a class
- Class Operations (Methods)
  - Operations are shown in the third partition. They are services the class provides.
  - The return type of a method is shown after the colon at the end of the method signature.
  - Without an object (i.e. an instance of a class), a method is non-existent
  - Class level method
    - –applies to class rather than an object of a class (aka static method). Underline it. e.g.–+findStudentsAboveHours(hours): Array
    - Class methods are not specific to any particular instance of a class (i.e. objects)

Design Class Diagram:

- Types of Classes:

  - Entity Class→Problem domain class that exists after the system is shut down
  - Boundary/View Class→Live on system's boundary→UI & Windows Classes
  - Controller Class→Mediates between boundary & entity classes, between view & domain layer.
  - Data Access Class→Retrieve & Send data to database
- 3 types of Visibility
  - Navigation visibility
    - Multiplicity is generally removed from the design class diagram to emphasise navigation from one object to an object
    - The ability of one object to view and interact with another object
    - add an object reference variable to a class.
    - Shown as an arrowhead on the association line
    - No multiplicity
  - Attribute visibility
    - Private (-): Attributes are usually designated as private.
    - Public (+): These attributes are accessed directly by another object.
    - Protected (#): These attributes are hidden from all classes except immediate subclasses
  - Method visibility
    - Public (+): A method can be invoked by another object
    - Private (-): A method can be invoked within class like a subroutine
- Different Cuts:
  - First-Cut→Define attribute & respective visibility
  - Final-Cut→Add methods

Cohesion & Coupling

- Cohesion→Measures the consistency of functions in a class
  - A single focus of class
  - A single focus of the methods in a class
  - No methods with multiple functions
  - Low cohesion (Tight):
    - Hard to maintain
    - Hard to reuse
    - Hard to understand
- Coupling→The extent classes knows about another (Loose V.S. Tight)
  - Loose → One interacts with another only through methods
  - Tight → One interacts with another through non-methods too
  - Shows dependency between classes as well
  - All classes should:
    - Be independent as far as possible
    - Low dependency/coupling
  - The qualitative measure of how closely classes are linked

Boundary class:

Boundary classes reflect the parts of the system that depend on its surroundings
THREE types of boundary class:

- Type 1 User interface classes
    - Classes involve communication with human users (Actors)
    - Examples: GUI like Windows, screens (including touch screens), menus, windows classes (e.g.: CustomerForm)
- Type 2 System interface classes
    - Classes involve communication with other systems
    - Examples: XML Files
- Type 3 Device interface classes
    - Classes that communicate with devices
    - Examples: Sensor class, printer class
- Thus, changing the GUI or XML file format means changing only the boundary classes, not the Entity and Control Classes

Controller class

- Controller class represents Objects that mediate between boundaries and entities class
- They serve as the glue between boundary class and entity class
- Controller class makes a system more tolerant of changes in the system boundary.

Communications among Classes

- Rule1: Actors can only talk to boundary objects.
- Rule 2: Boundary objects can only talk to controllers and actors.
- Rule 3: Entity objects can only talk to controllers.
- Rule 4: Controllers can talk to boundary objects and entity objects, and to other controllers, but not to actors

The System Sequence Diagram (SSD)

- Definition:
    - Shows interactions between an ACTOR and ONE OBJECT in a SINGLE USE CASE
    - The one object represents the COMPLETE SYSTEM (represented as:System).
    - Time runs from top to bottom.
    - Inner workings of System that are not immediately visible
- Components of SSD: Actor, :System, object lifeline, Messages
- The vertical line under object or actor:
    - Shows passage of time
    - The Life-Line represents the object's life during the interaction
- Activation Box (Long narrow rectangles)
    - Indicate that object is ACTIVE only during part of scenario (i.e. receiving or sending messages)
    - Otherwise, the object is in IDLE state
- Return messages are indicated using a dashed arrow with a label indicating the return value. It's OPTIONAL when nothing is returned

<u>Developing a System Sequence Diagram</u>

- Identify input messages using
  - activity diagrams
  - use case descriptions
- Describe message from External Actor to System using message notation
  - Name it verb-noun: what the system is asked to do
  - Consider parameters the system will need
- Identify and add any special conditions on the input message
  - – Iteration/loop frame
  - – Opt or Alt frame
- Identify and add output return messages

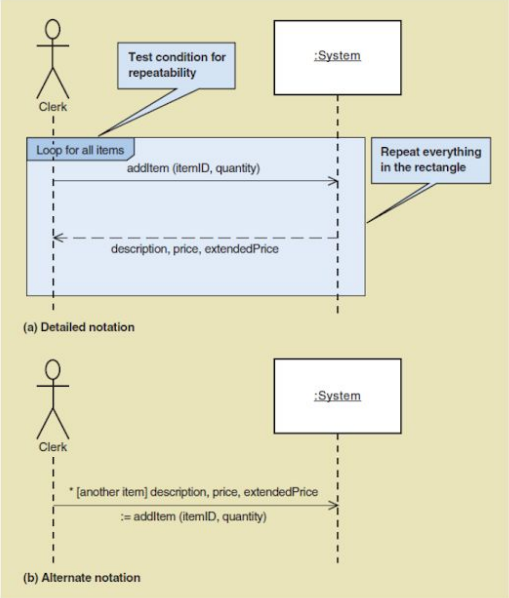


Fig 1: SSD Message with Loop Frame

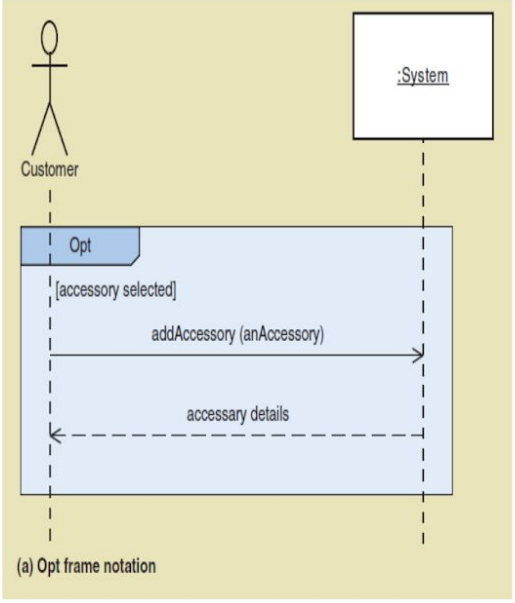


Fig 2: SSD Message Examples: Opt Frame (optional)

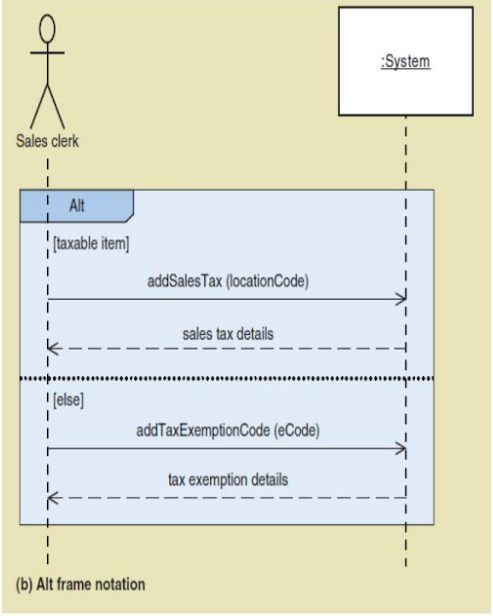


Fig 3: SSD Message Examples: Alt Frame (if-else)

- – On message itself: aValue:= getValue(valueID)
- – As an explicit return on the separate dashed line

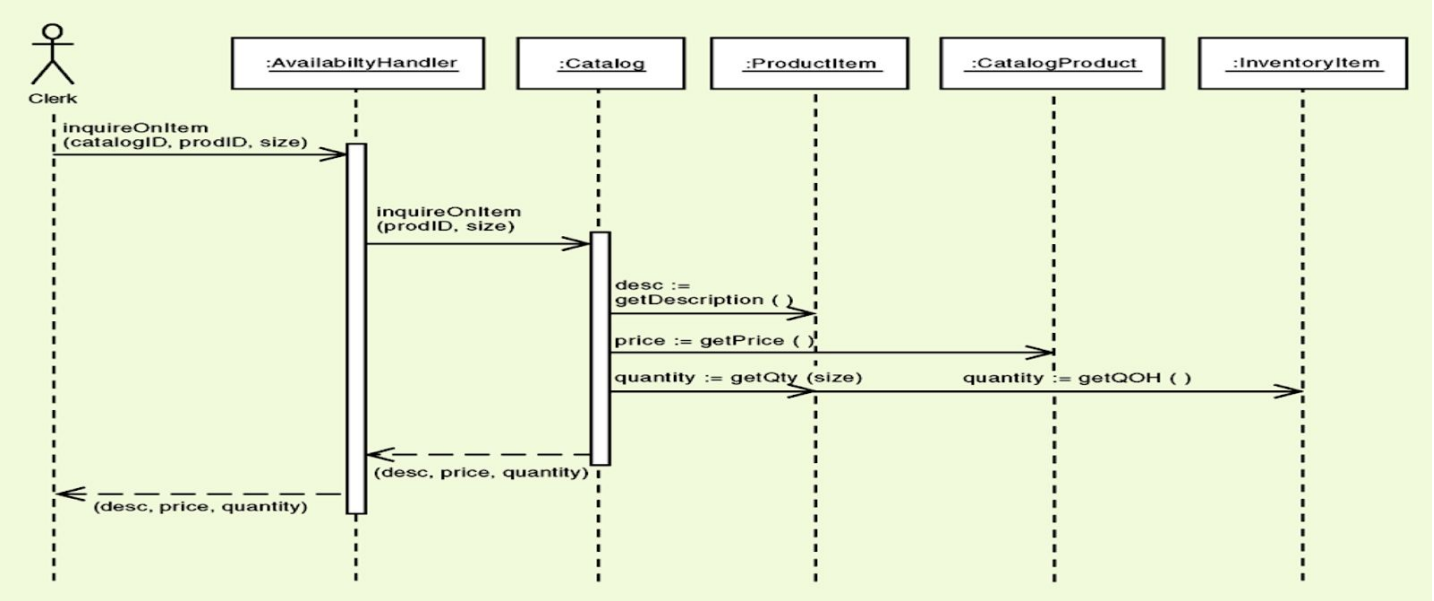
Sequence diagram

- Definition:
  - A sequence diagram is used in Systems DESIGN
  - Shows the design class objects that participate in a use case and messages (interactions) among them
- Sequence diagram represents a 2-dimensional graph
  - Objects are shown in the horizontal dimension
  - Sequence of messages is shown in the vertical dimension(from top to bottom)

<u>Develop First-cut & Final-cut sequence diagram:</u>

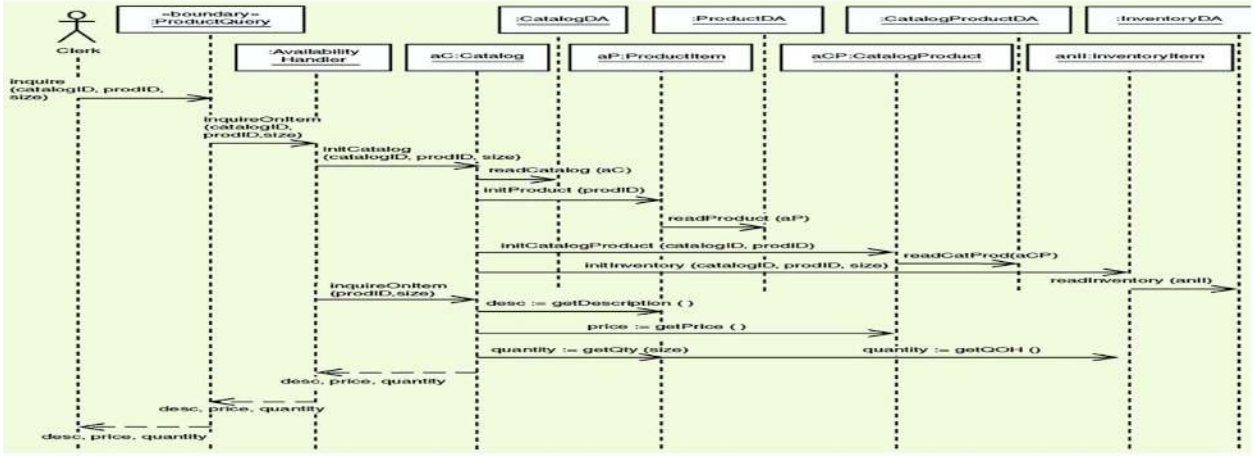
- Start with elements from SSD
- Step 1: Replace the: System object with an appropriately named use case controller
- Step 2: Add other objects to be included in the use case
- Step 3: Select the input message from the use case

- Step 4: add all objects that must collaborate by providing or obtaining data (consult design class: see next slide)



Final cut sequence diagrams:

- Represent a three-layer architectural design:
  - view layer
  - problem domain layer
  - data layer
- Step 1: add a view layer interface class before the controller either as a single GUI class or as Windows classes
- Step 2: add a data access class for each problem domain class



Sequence diagram VS System sequence diagram

Sequence diagram	System sequence diagram
The elements participating in a sequence	The elements participating (exchanging

diagram are objects (instances of various classes).	messages) in a system sequence diagram are Actors and Systems
It shows the details of what happens inside a system	SSD is actually a sub-type of sequence diagram made at the highest level
The sequence diagram is developed as a key activity of systems design	Elements of SSD should be described in user terms, without IT slang. Because it belongs to the requirements part of the project documentation

Seminar 10: Security & Testing

System Security:
Ensuring Reliability & Protection from Risks

- 6 Types of Risks:
  - i. Human Error
  - ii. Technical Error
  - iii. Accidents & Disasters
  - iv. Fraud
  - v. Commercial Espionage
  - vi. Malicious Damage

Information Security:

- CIA Model:
- i. Confidentiality:
  - 1. Allows authorised users to view & process data only.
  - 2. Ensure no disclosure to unauthorised people/systems.
  - 3. Data encryption is used.
- ii. Integrity:
  - 1. Prevents data modification from unauthorised parties.
  - 2. Prevents data modification while in transit over the Internet.
  - 3. Alerts management upon attempts made by unauthorised personnel.
- iii. Availability:
  - 1. Ensures timely & accurate access to data.
  - 2. Mishap/Malice cannot make systems unavailable.
  - 3. Protection from Cyber Attacks

How to keep IS secure:

- Prevention of errors and breaches
- Detection to spot security breaches
- Deterrence to discourage breaches
- Data recovery to recover lost data or information

Testing:
<u>Why do we need testing?</u>

- The primary aim of testing is always to get the system to fail (to find errors)
- NOT able to produce 100% error-free software no matter what.
- These errors led to systems failure
  - Expensive
  - Dangerous
  -

<u>Test Process</u>

- Plan - “What” to test
- Design - “How” you are going to test

- Schedule - “When” you are going to test
- Execute - “What was the result?”

Testing process in different development approaches

- Waterfall
  - Generally all testing and quality control points come late in the project (if time permits)
  - Testing is usually scheduled late because people think that testing can only be done on code – NOT TRUE
- Agile
  - In agile development, testing is integrated throughout the lifecycle; testing the software continuously throughout its development.
  - NO a separate test phase
  - Developers heavily engaged in testing, writing automated repeatable unit tests to validate their code.
  - Supports the principle of small, iterative, incremental releases.
  - Testing is done as part of the build, ensuring that all features are working correctly each time the build is produced.
  - Integration is done as you go.
  - The purpose of these principles is to keep the software in releasable condition throughout the development, so it can be shipped whenever it's appropriate.

Test Methods

- Black Box Testing
  - – Testing, either functional or non-functional, without reference to the internal structure of the component or system (i.e. code not visible)
  - – Typically executed in functional test phases i.e. Unit, Integration, System or Acceptance test phases
  - – Can be independently tested
- White Box Testing
  - – Uses an internal perspective of the system to design test cases based on internal structure.
  - – It requires programming skills to identify all paths through the software
  - – Due to internal perspective, maximum coverage of a scenario is possible
- – Grey Box Testing
  - – Is a combination of black box and white box testing
  - – We look into the “box” being tested just long enough to understand how it has been implemented. Then we close up the box and use our knowledge to choose more effective black box tests.
  - – Increase testing coverage

	Black Box Testing	Grey Box Testing	White Box Testing
1.	The Internal Workings of an application are not required to be known	Somewhat knowledge of the internal workings are known	Tester has full knowledge of the Internal workings of the application
2.	Also known as closed box testing, data driven testing and functional testing	Another term for grey box testing is translucent testing as the tester has limited knowledge of the insides of the application	Also known as clear box testing, structural testing or code based testing
3.	Performed by end users and also by testers and developers	Performed by end users and also by testers and developers	Normally done by testers and developers
4.	-Testing is based on external expectations -Internal behavior of the application is unknown	Testing is done on the basis of high level database diagrams and data flow diagrams	Internal workings are fully known and the tester can design test data accordingly
5.	This is the least time consuming and exhaustive	Partly time consuming and exhaustive	The most exhaustive and time consuming type of testing
6.	Not suited to algorithm testing	Not suited to algorithm testing	Suited for algorithm testing
7.	This can only be done by trial and error method	Data domains and Internal boundaries can be tested, if known	Data domains and Internal boundaries can be better tested

Testing Types

- Functional Testing: Testing whereby the system is tested against the requirements specification
    - Unit, Integration, System, Acceptance
- Regression Testing
- Static & Dynamic Testing
- Performance, Load & Stress testing – Usability testing
- Accessibility testing
- Security testing
- Backup & Recovery testing

Unit Testing :

- To test individual methods/classes/components before integration.
- By developers themselves.
- Advantage:
    - Errors are found at early stage → errors can be resolved, without impacting the other piece of code
    - Reducing the cost of bug fixes at a later stage. Bugs detected at later stages of software development are expensive and time-consuming to fix

Integration Testing :

- Tests the interaction & consistency of an integrated system.
- Interfaces between modules.
- Looking for:
    - Unexpected Parameter values

- Run-time exceptions
- Unexpected state interactions

System Testing:

- Tests the complete system
- Integration test of an entire system or independent subsystem at the end of each iteration

Performance test or stress test:

- Seeks to determine whether a system or subsystem can meet time-based performance criteria
- E.g. Response Time/Throughput

User acceptance test:

- To determine whether a software system fulfils user requirements
- near the end of the project
- By END-USERS ONLY
- Plan the UAT
  - be done early in the project
  - Test cases for every use case and user stories
  - Identify conditions to verify that the system supports the use case accurately and completely
  - Document and track results (especially errors and fixes)

Regression Testing

- Testing of a previously tested program following modification to ensure that defects have not been introduced or uncovered in unchanged areas of the software, as a result of the changes made.
- Can be performed in all test phases

Static Testing

- is a form of software testing where the software isn't actually used.
- not detailed testing, but checks mainly for the sanity of the code, algorithm, or document.
- It is primarily syntax checking of the code or manually reading of the code or document to find errors.
- This type of testing would primarily be used by the developer who wrote the code, in isolation.

Accessibility Testing

Testing to determine the ease by which users with disabilities can use a component or system.

Usability Testing

- Testing to determine the extent to which the software product is understood, easy to learn, easy to operate and attractive to the users under specified conditions. [After ISO 9126]
- Typically executed in the later phases of testing when system is more stable i.e. System Test

Security Testing

- The process to determine that a System protects data and maintains functionality as intended. The six basic security concepts that need to be covered by security testing are: confidentiality, integrity, authentication, authorisation, availability and non-repudiation.

Backup & Recovery Testing

- Taking a backup is the process of taking a copy of an application, its data and environment, such that it may need to be recovered (or restored) in the event of a system failure
- Recovery is the process of reverting to the backup copy of an application, its data and environment such that normal business can be resumed.
- Backup & recovery testing is disruptive so is therefore executed during the quiet times of the later phases of testing

Test Preparation Principles

- Starts early in the project lifecycle
- Produce a Test Strategy Plan first
- Review all documentation
    - Business Requirements
    - FunctionalRequirements
- Design test requirements/conditions
- Design test cases (manual test) or scripts (automated test) to meet the test requirements
- Prepare the test environment(s) and required data

Test plan

- Testing needs to be planned in detail
    - how and when testing will be done, how long it will take
    - types of tests, things to be tested/not tested
    - resources, staff, facilities, tools, training needed
    - schedule
    - risks

Test case

- a formal description of a starting state, one or more events to which the software must respond, and the expected response or ending state
- Is defined based on:
    - well understood functional & non-functional requirements

- Must test all normal and exception situations

Automated testing tools

- Document and create test cases
- Automated running of test cases
  - – Including GUI scripts
  - – Load testing (might run on hundreds of client machines to test typical load)
- Recording of results
  - Comparison of actual to expected results
  - Comparison of different test trials

Seminar 11: Implementing & Maintaining the System

Implementing the System

1. ImplementationPlanning
    a. Review Acceptance Checklist, Prepare Implementation Schedule
2. Build the System or Buy the System
    a. Will result in a varied implementation path
3. Test the System (covered in Week 10)
    a. SystemTesting(functional and performance), Acceptance Testing
4. Finalise documentation
    a. System Documentation,User documentation
5. Get ready for the System to go Into Production
    a. Data conversion/migration,Configure The Production environment, Conduct training
6. Deploy The System
    a. Install/deploy the system,Monitor operations, Benchmark testing, Tune the system,
7. Wrapup
    a. Operations Handover,Transition Support, SystemClosure, Post-implementation review

Building V.S. Buying:

Building (In-House Development)→Assembling an internal IT project team that supports the software in the long run, whilst meeting business requirements.

- Advantages
    - Customisation is easy due to familiarity
    - Can develop and train IT staff with new skills
- Drawbacks
    - However, takes greater time for Developing, Testing & Upgrading

Buying→Purchasing standard software packages commercially available from Software Vendors/Value-Added Reseller,

- i. Horizontal Applications→Used in many types of organisations
- ii. Vertical Applications→Customised to meet requirements of a specific type of business
- Why we buy?:
    - Reason 1: When an IT application is needed for a standard business function
    - Reason 2: When resources are in short supply for in-house development
    - Reason 3: When the BUY option is more cost-effective than in-house development
    - Reason 4: a system needs to be available in a short space of time
    - Reason 5: Because the system and documentation are usually maintained by the vendor, no internal development is necessary
    - Reason 6: Upgrades are done by the vendor
    - Reason 7: Software package is of HIGH QUALITY

    - Reason 8: Political
- Risk:
    - Very rare to find a package that can do everything well that a user wants
    - Often need to develop specialised package additions because the multi-purpose packages do not handle certain functions well
    - Conversion and integration costs can sometimes be so significant as to render the project infeasible
    - Some vendors refuse to support packages which have been customised by the users .. and most packages need some customisation
    - Customisation can be so extensive that it would have been cheaper to develop the system in-house
- Build vs. Buy considerations
    - Functionality
    - Cost
    - Vendor Support
    - Viability of Vendor
    - Widespread use by others(client references) • Flexibility
    - Documentation
    - Performance and scalability
    - Ease of Installation

IT outsourcing:

- It refers to the process of subcontracting the IT function of an organisation to a third-party
- governed by a LEGAL CONTRACT:
    - the services that a vendor is to provide;
    - And how the service delivery is to be measured
    - Penalties for non-performance
    - discusses financial arrangement;
    - Indicates Legal Issues
- Benefit:
    - Reduced costs: Economics of scale
    - Strategic: focus on core business activities & allocate more resources to core business activities
    - Access: To the latest ICT skills and technologies
    - Increased share price: favourable market reaction
- Disadvantage
    - Loss of distinctive competencies: others learn ideas
    - Privacy concerns: no strongly enforces in some countries
    - Increased vendor dependency: monopoly

System Documentation: Used to facilitate communication during development

User Documentation: Description of how to use the system for end users

Data conversion / migration: Process of getting data ready for the new system

Configure the production environment: Ensure all facilities are set up

Conduct Training:

- Considerations:
  - Users
  - Number of users, Existing skill levels, On-going usage levels – regular, occasional
  - Level Of Detail To Be Imparted To The Audience
  - Who should conduct the training
  - Where/When Should The Training Be Conducted
  - Methods and resources, specialised training documentation– designed to put novice users at ease
  - Need supportive User Manager who committed allocating time for training
- Training aids:
  - Must Be Easy To Use
  - Reliable
  - Demonstrations And Classes
  - Training documentation, especially designed to put the novice user at ease
  - On-line help
  - Expert users
- On-going training needs after installation:
  - Onlinehelp
  - Resident Experts
  - Helpdesk

Four Deployment Approaches:

- Direct deployment
- Parallel deployment
- Pilot deployment
- Phased deployment

<u>Direct deployment:</u>

- Installs a new system, quickly makes it operational, and immediately turns off any overlapping OLD system
- Advantage: Lowest cost because two systems are not to be maintained in parallel
- Suitability:
  - system failure is not critical
  - prefer direct deployment
  - systems built in-house for non-critical systems
  - The environment cannot support both the old and the new systems
- Risks:
  - Higher risk because, if something goes wrong, reverting back to the old system usually is difficult
  - A system usually encounters difficulties because live data (e.g. sales transaction data) typically occurs in much larger volumes than test data

- Detecting and understanding minor errors are hard → users cannot verify current output by comparing it to output from the old system

Parallel deployment:

- The new system & old system are kept running in parallel (side-by-side) for a while
- Data that are input into the old system and new one, and then compared
- when the new system is found to be correctly working, the old system will be completely withdrawn
- Advantages:
  - If anything goes wrong with the new system, the old system will act as a back-up, until appropriate changes are made in the new system
  - The outputs from the old and new systems can be compared to check the performance of the new system
- Problems:
  - Inputs (data, format, and frequency) may differ
  - Entering data into two systems takes a lot of extra time and effort
  - HIGHER COST.
  - Heavier load on equipment
  - Extra workload on staff
  - Could cause processing delays

Pilot deployment:

- The completely new system (involving all functions) is piloted (trialled) at a selected location (e.g. in one department, in one branch) of an organisation till new system is fully working
- Advantages:
  - All features of the new system can be fully tested in a work environment
  - If something goes wrong with the new system, only a small part of the organisation is NOT even affected
  - The staff who were part of the pilot scheme can help train other staff
  - less expensive than a parallel operation
- Disadvantage:
  - A bit costly than the direct approach

Phased deployment:

- The new system is introduced in phases (stages, or steps). When a phase is successful, the next phase begins → gradually replaces parts of the old system until the new system fully replaces the old system
- Advantages:
  - The risk of errors or failures is limited to the implemented module only
  - Allows users to gradually get used to the new system
  - Lower risk
  - Staff training can be done in stages
  - Less expensive than a full parallel deployment approach
- Disadvantages:

- If a part of the new system fails, there is no back-up system, so data can be lost
- Slower than direct implementation
- The client must wait until all the phases are complete to evaluate the whole system.

FOUR types of maintenance

- Corrective → Initiated by BUG reports: 1. Focus on removing defects
- Adaptive→Making the software/system respond to new environment/conditions
- Perfective→Meet user requirements not previously recognised/prioritised
- Preventative→Fixing potential problems noticed while fixing something else

Post-Implementation Review (PIR)

- A PIR analyses what went right and wrong with an IT project
- It is conducted 2 to 6 months after deployment
- CONCERN 1:
  - look at original requirements and evaluate how well they were met
- CONCERN 2:
  - compare costs of development and operation against original estimates
- CONCERN 3:
  - compare original and actual benefits
- CONCERN 4:
  - new system reviewed to see whether more of original or additional benefits can be realised