

VU LEARNING HUB

vulearninghub.xyz

HIGHLIGHTED HANDOUT

SE601

Software Construction & Development

SE601 by VU Learning HUB

Prepared for VU students • Free study resources

<https://vulearninghub.xyz>

Software Construction & Development

Topic 1

What is Construction?

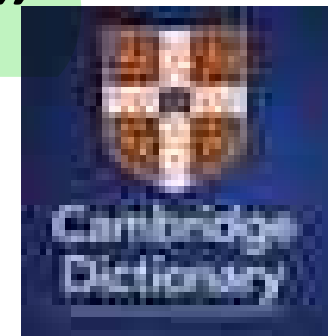
What is Construction?

Construction Noun

“the **process**, **art**, or **manner** of **building** something”



“the **work** of **building** or **making** something”



Software Engineering Body of Knowledge (SWEBOK)

major Knowledge Areas?

Software requirements

Software design

Software construction

Software testing

Software maintenance

ISO/IEC TR 19759



What is Software Construction?

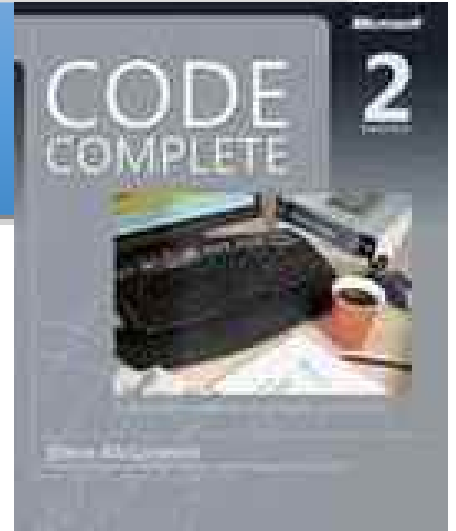
What is Software Construction?

The term software construction refers to the detailed creation of working software through a combination of coding, verification, unit testing, integration testing, and debugging. (SWEBOK)



What is Software Construction?

Creating software through
combination of detailed design,
construction planning, coding,
debugging, unit testing, integration
testing, and considering corrective
maintenance.



Steve McConnell
Winner of Software Development
Productivity Award

Code Complete: A Practical Handbook of Software Construction. By Steve McConnell, Second Edition, Microsoft Press, 2019.



Software Construction: objective

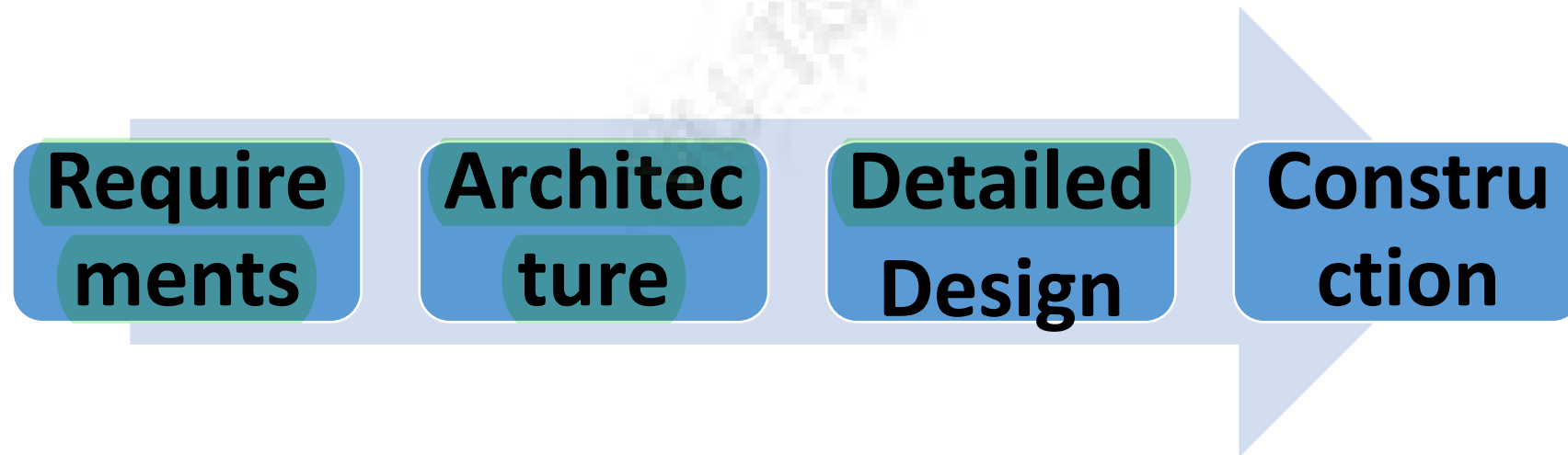
- Objective is to map **detailed design** on to **correct** and **manageable** code

Code Complete: A Practical Handbook of Software Construction. By Steve McConnell, Second Edition, Microsoft Press, 2019.



Software Construction: when to start

- Software Construction is an activity of software development started after **requirement engineering**, **architecture** and **detailed design**

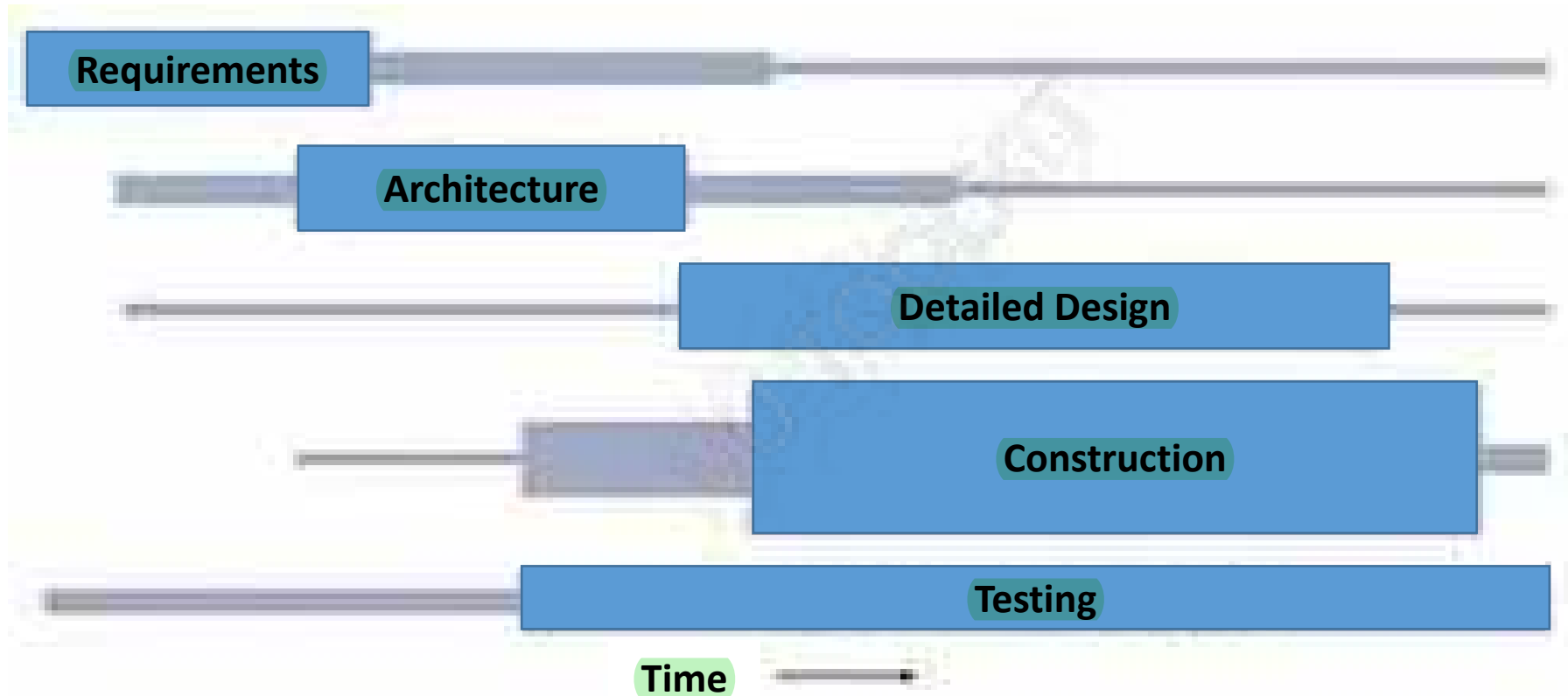


Code Complete: A Practical Handbook of Software Construction. By Steve McConnell, Second Edition, Microsoft Press, 2019.



Overlapping Software Construction with Other Activities

Activities overlapped to some degree on most projects



Code Complete: A Practical Handbook of Software Construction. By Steve McConnell, Second Edition, Microsoft Press, 2019.



Software Construction Activities

Software Construction Activities

Topic 2

Software Construction Activities

The major activities in construction are

Coding

debugging

integration

developer testing (unit testing and integration testing)

and continuous liaison with **detailed design**.

Software Construction Activities



Construction Activities (inside circle) focuses on coding and debugging but also includes detailed design, unit testing, integration testing, construction planning and other activities

Specific Tasks involved in Software Construction Activities

1. Designing and writing **classes**, methods and **relationships**
2. Creating and **naming variables** and **named constants**
3. Selecting **control structures** and organizing blocks of statements
4. **Polishing code** by carefully formatting and **commenting**
5. Code **safety, Reliability** and **ethics**
6. Ensure compliance to good **Software Coding Standard**

e.g. MISRA C++

Specific Tasks involved in Software Construction Activities

7. **Verifying** that the code build according to specification
8. Determining **how** your code will be **tested**
9. **Integrating** software components that were created separately
10. **Tuning** code to make it **faster** and use **fewer resources**

Problem Definition, Requirement Development and Construction Planning

TOPIC 3

Major Software Development Activities

- **Problem definition**
- **Requirements development**
- **Construction planning**
- Software architecture, or high-level design
- Detailed design
- Coding and debugging
- Unit testing
- Integration testing
- Integration
- System testing
- Corrective maintenance

Problem Definition

Considered to be prerequisite of Software Construction

Clear statement of the problem that the system is supposed to solve

Sometimes called “**Product Vision**” Or “**mission Statement**”

Good and Poor Problem Statement

A problem definition defines what the problem is without any reference to possible solutions.

Example 1: We can not keep up with orders for the IBM (Good)

Example 2: We need to introduce Blockchain based distributed digital ledger system to keep up with orders for the IBM (Poor)

It doesn't sound like a problem; it sounds like a solution

Problem definition as Foundation of Development Process



Requirement Development

Requirements are statements of what the system **must do**, how it **must behave**, the **properties it must exhibit**, the **qualities it must possess**, and the **constraints** that the system and its development must satisfy.

SEI.CMU

Managing Construction

- **Construction in Life Cycle Models**

Linear vs Iterative models

- **Construction Planning**

Order in which software components are created, allocation of tasks, quality

- **Construction Measurement**

code developed, code modified, code reused, code destroyed, code complexity,

code inspection statistics, fault-fix and fault-find rates, effort, and scheduling.

Software Architecture, High-Level and Detailed Design

TOPIC 4

Major Software Development Activities

- Problem definition
- Requirements development
- Construction planning
- **Software architecture, or high-level design**
- **Detailed design**
- Coding and debugging
- Unit testing
- Integration testing
- Integration
- System testing
- Corrective maintenance

Software Architecture

“the set of structures needed to reason about the system, which comprise software elements, relations and properties” SWEBOK

Big Picture : What, Where, but not how

Software Architecture Vs High Level Design

Architecture refers to design constraints that apply **system wide**

whereas

High-level design refers to design constraints that apply at the **subsystem** or multiple-class level, but not necessarily system wide (**SWEBOK**)

Architecture is also known as "system architecture," "high-level design," and "top-level design."

Detailed Design

Refines the elements of the **software architecture** and provides a **detailed blueprint** to the programmers responsible for the implementation of the software

e.g. class diagrams, ER model, sequence diagrams, activity diagrams, naming conventions, data storage specification, I/O format

Reverse and Forward Engineering Principle

Software Structure and Architecture

1-Architectural Structures and Viewpoints

2- Architectural Styles

3- Design Patterns

4- Architecture Design Decisions

5-Families of Programs and Frameworks

Software Structure and Architecture

1-Architectural Structures and Viewpoints

(logical view, Process view, Physical view, deployment view)

Software Structure and Architecture

2- Architectural Styles

- General Structure (Layers, pipes, filter)
- Distributed Systems (client/server, three-tiers, broker)
- Interactive Systems (microkernel, reflection)
- Adaptable (batch, interpreters, process control, rule based)
- Peer to Peer e.g. blockchain

Software Structure and Architecture

3- Design Patterns

- Creational Patterns (Factory, Prototype, Builder, Singleton)
- Structural Patterns (Adaptor, Façade, Proxy)
- Behavioral Patterns (observer, state, iterator)

Software Structure and Architecture

4- Architecture Design Decisions

- Trade-off among quality attributes e.g Time-Space trade Off, sentinel Approach

5-Families of Programs and Frameworks

- designing reusable and customizable components

Software Structure and Architecture

5-Families of Programs and Frameworks

- designing reusable and customizable components

Coding and Debugging

Topic 6

Major Software Development Activities

- Problem definition
- Requirements development
- Construction planning
- Software architecture, or high-level design
- Detailed design
- **Coding and debugging**
- Unit testing
- Integration testing
- Integration
- System testing
- Corrective maintenance

Coding

Coding is an activity of writing instruction in some programming language

“software construction is mostly coding and debugging”

SWEBOK

Expertise required for Coding

- knowledge of the application domain
- appropriate data structures
- specialized algorithms
- language constructs
- good programming practices
- Knowledge of Coding Standards

What is Bug?

a bug is an **error** in the
source code that causes a
program to produce
unexpected results or **crash**



What is Debugging?

“Debugging is the activity to find and fix bugs (faults) in the source code (or design)” (SWEBOK)

Debugging Techniques

- **Static Debugging (Code Review)**
- **Dynamic Debugging (closely associated with testing)**
- **Postmortem Debugging (Core dump of a process)**

Unit and Integration Testing

Topic 06

Major Software Development Activities

- Problem definition
- Requirements development
- Construction planning
- Software architecture, or high-level design
- Detailed design
- Coding and debugging
- **Unit testing**
- **Integration testing**
- Integration
- System testing
- Corrective maintenance

What is Unit Testing ?

Software Construction Activities



Construction Activities (inside circle) focuses on coding and debugging but also includes detailed design, unit testing, integration testing, construction planning and other activities

Unit Testing (Activity of Software Construction)

- Testing of **smallest piece of code** that can be logically

isolated

e.g. class, function, component

- Carried out by **developers**

Goal: standalone module is functionally **correct** and **fit for use**

Example : Binary Search (Unit test)

```
bool BinarySearch (A, key, low, high)
```

```
mid ← (low + high)/2
```

```
While ( key!=a[mid]) and ( low<high)
```

```
    if (key >a[mid])
```

```
        low= mid +1;
```

```
    else
```

```
        high = mid-1
```

```
        mid ← (low + high)/2
```

```
If ( low <high )
```

```
    return true
```

```
else
```

```
return false
```

Example : Binary Search (Unit test)

Array



0 1 2 3 4 5 6 7 8 9

Low

high

bool BinarySearch (**A, key, low, high**)

Example : Binary Search (Possible Inputs and Expected Output)

Possible Test Case	Input	Expected Output
1.Array of size 1, element in array	[10] key=10	true
2.Array of size 1, element not in Array	[20] key=10	false
3.Even Array size , 1st element in Array	[10,20,30,40] key=10	true
4.Even Array size , Last element in Array	[10,20,30,40] key=40	true
5.Even Array size , not first or last element in Array	[10,20,30,40] key=50	false
6.Even Array size , element not in Array		
7.Odd Array size , 1st element in Array		
8.Odd Array size , Last element in Array		
9.Odd Array size , not first or last element in Array		
10.Odd Array size , element not in Array		

Example : Binary Search (Unit test case in C++)

```
bool UnitTestCase ( ){
```

```
    bool flag =true; int *a; int key=10;
```

```
    a= new int [1]; a[0]=key;
```

```
    if ( BinarySearch(a,key,0,1)!= true) // Possible Test Case 1
```

```
        flag=false;
```

```
    a[0]=20;
```

```
    if ( BinarySearch(a,key,0,1)!=false) // Possible Test Case 2
```

```
        flag=false;
```

```
    delete []a; a= new int[4]; a[0]=key, a[1]=20, a[2]=30;a[3]=40;
```

```
    if (BinarySearch(a,key,0,4)!=true) // Possible Test Case 3
```

```
        flag=false;
```

```
    .....
```

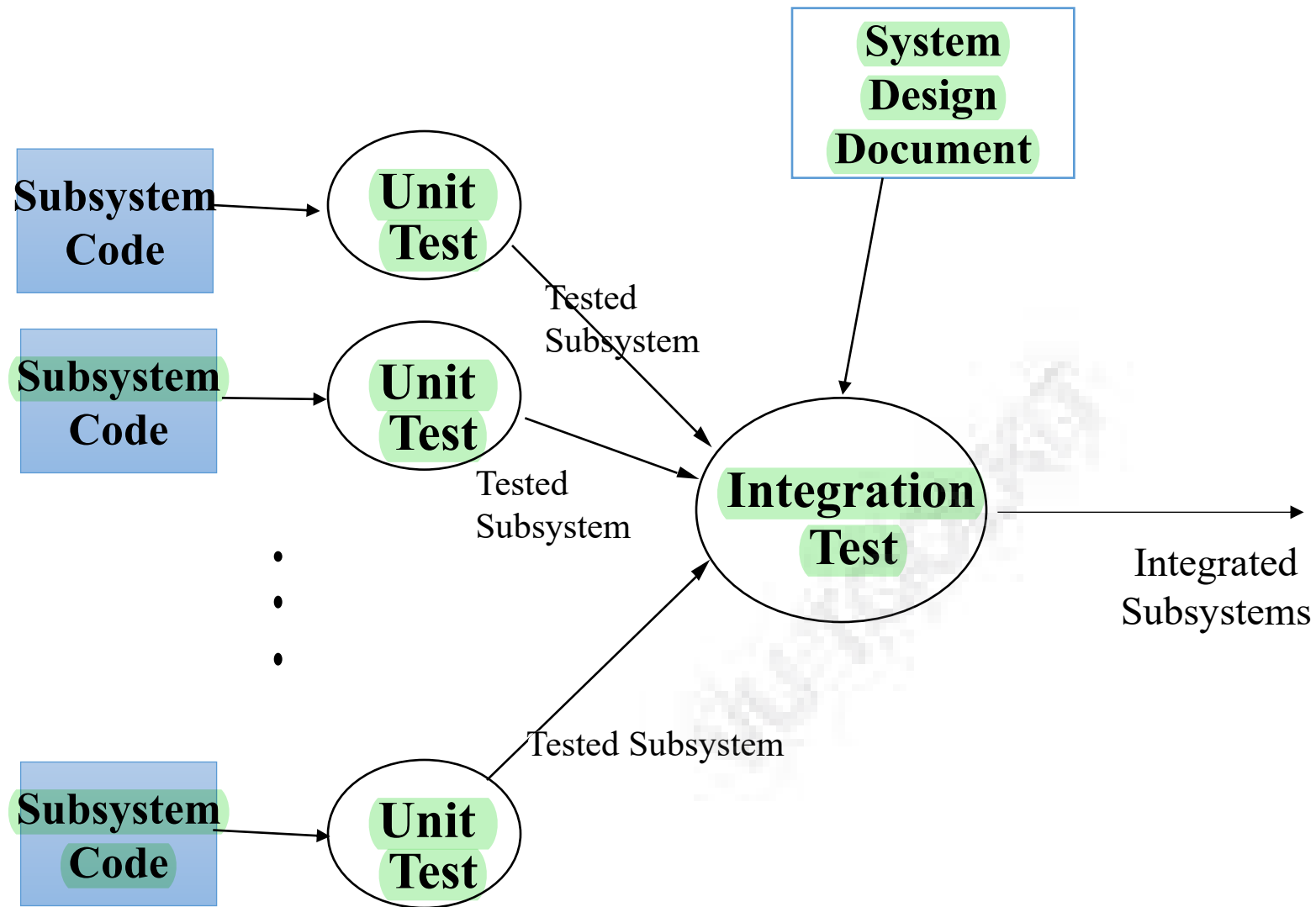
```
    return flag
```

```
}
```

Integration Testing (Activity of Software Construction)

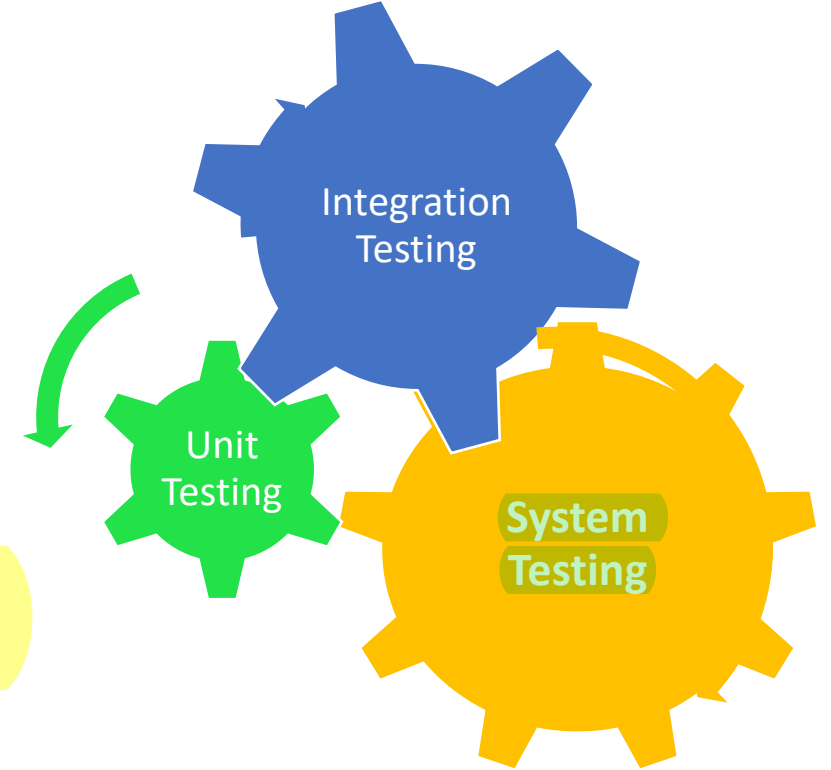
- **Testing groups of subsystems** (collection of classes, packages, components create by multiple programmer or teams)
- **Carried out by developers**

Goal: Test the interface among the subsystem



All tests by developer

System Testing Topic



Major Software Development Activities

- Problem definition
- Requirements development
- Construction planning
- Software architecture, or high-level design
- Detailed design
- Coding and debugging
- Unit testing
- Integration testing
- **System testing**
- Corrective maintenance

System Testing

- test **entire system behavior** as a whole, with respect to **scenarios** and **requirements**
- execution of the software in its **final configuration**
- tests for **security, performance, scalability, reliability, security** and other issues

Importance of System Testing



Importance of System Testing



Apple Engineer died in Tesla Crash Had Previously Complained About Autopilot

Source: *Credit: Eric Marrapodi/CNN)*

Importance of System Testing



Software bug led to death of pedestrian in Uber's self-driving crash



Importance of System Testing



Malfunction due to software, leading to 3 dead and 3 critically injured

Therac-25 radiation machine used to treat cancer patients to remove tumor

Source: <https://kalfaoglu.com>



Importance of System Testing

Failed Satellite launch – 1.2 \$ billion Lost

Source: NASA

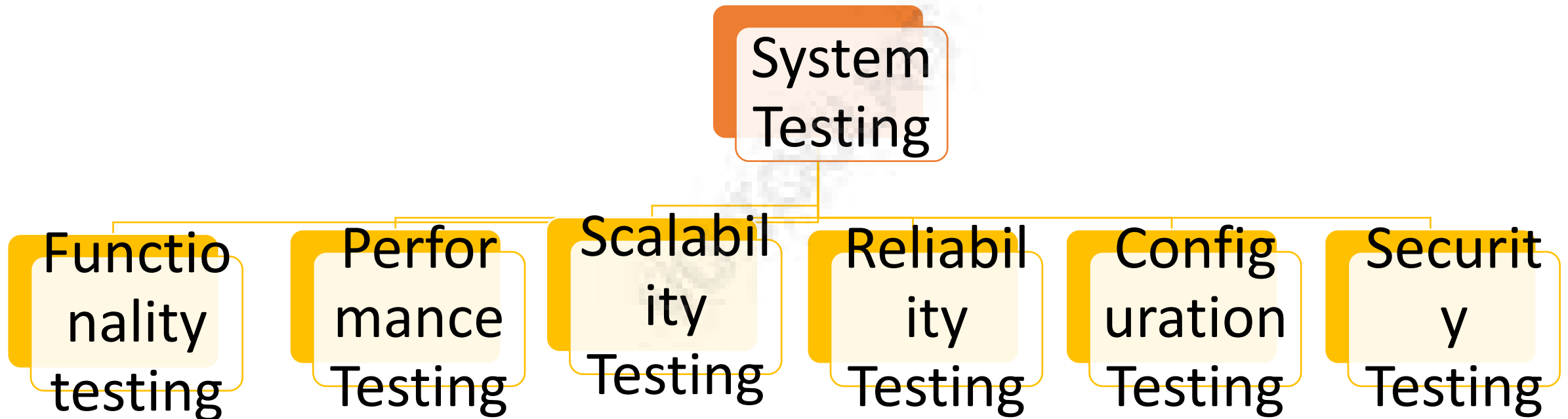


Importance of System Testing



Public launch of Windows 98 Fails

System Testing Types



Software Construction & Development

Week-2

Software development Process

- In software engineering, a software development process is a process of dividing software development work into smaller, parallel, or sequential steps or sub-processes to improve design and/or product management.
- Provides a roadmap to software development
- Types of Software development Processes
 - Conventional
 - Agile

Steps in the software development process



<https://www.spaceotechnologies.com/blog/software-development-process/>

Software Development Process Phase 1

1. Requirement Gathering & Analysis Phase

Goal: Document functional and non-functional requirements(NFRs).

Who: Business Analyst, Functional/Business Users, Architect(s), Lead Developer, Project Manager or Scrum Master.

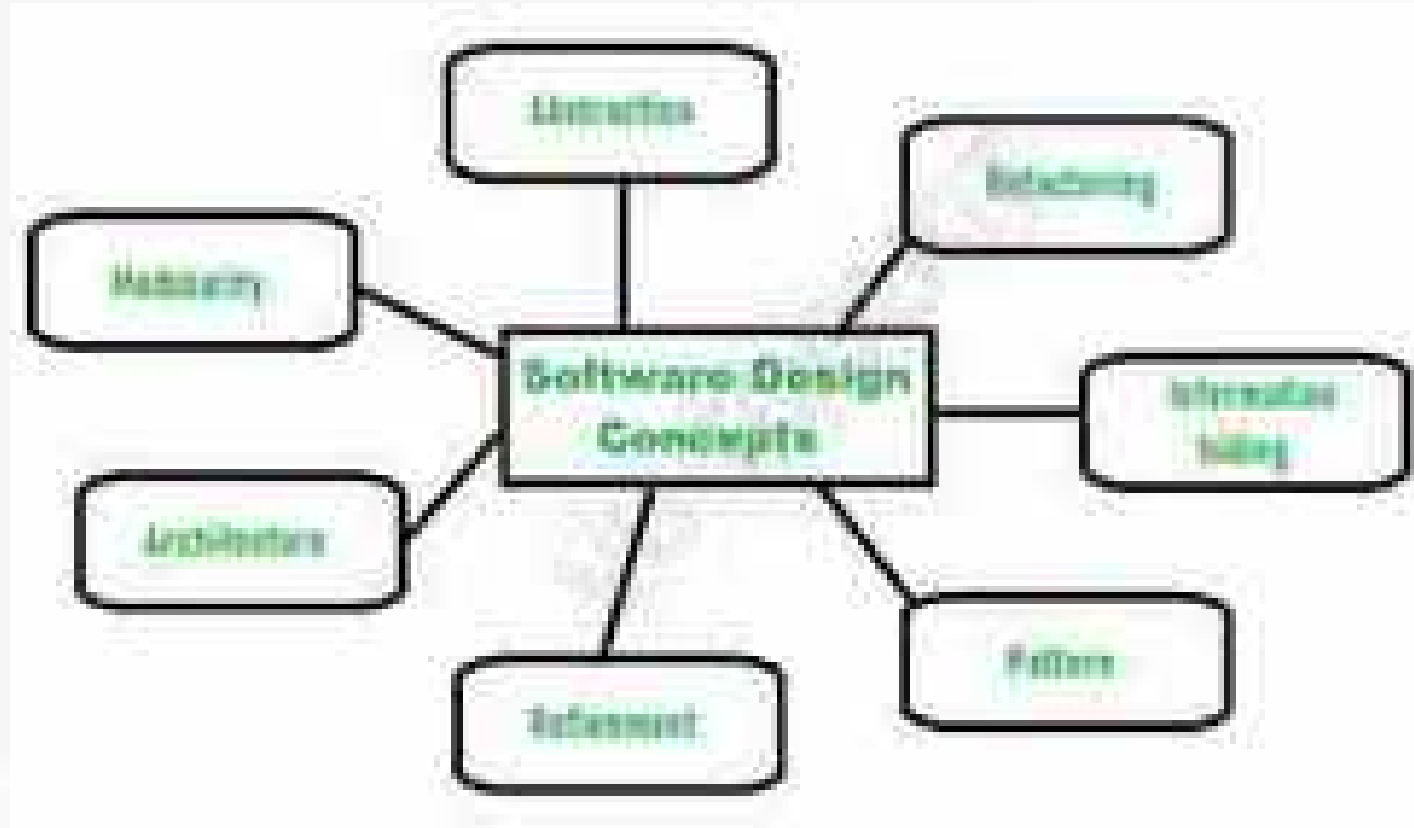
Process: Stakeholder discussions, brainstorming sessions, interviews, Prototyping, Role playing, Review existing system/process.

Deliverables: Identified Actors, UseCases, Screen Prototypes, UX designs, Reviewed and Approved requirements.

-by SFSupport247.com



Software Design and Prototyping



<https://www.geeksforgeeks.org/introduction-of-software-design-process-set-2/>

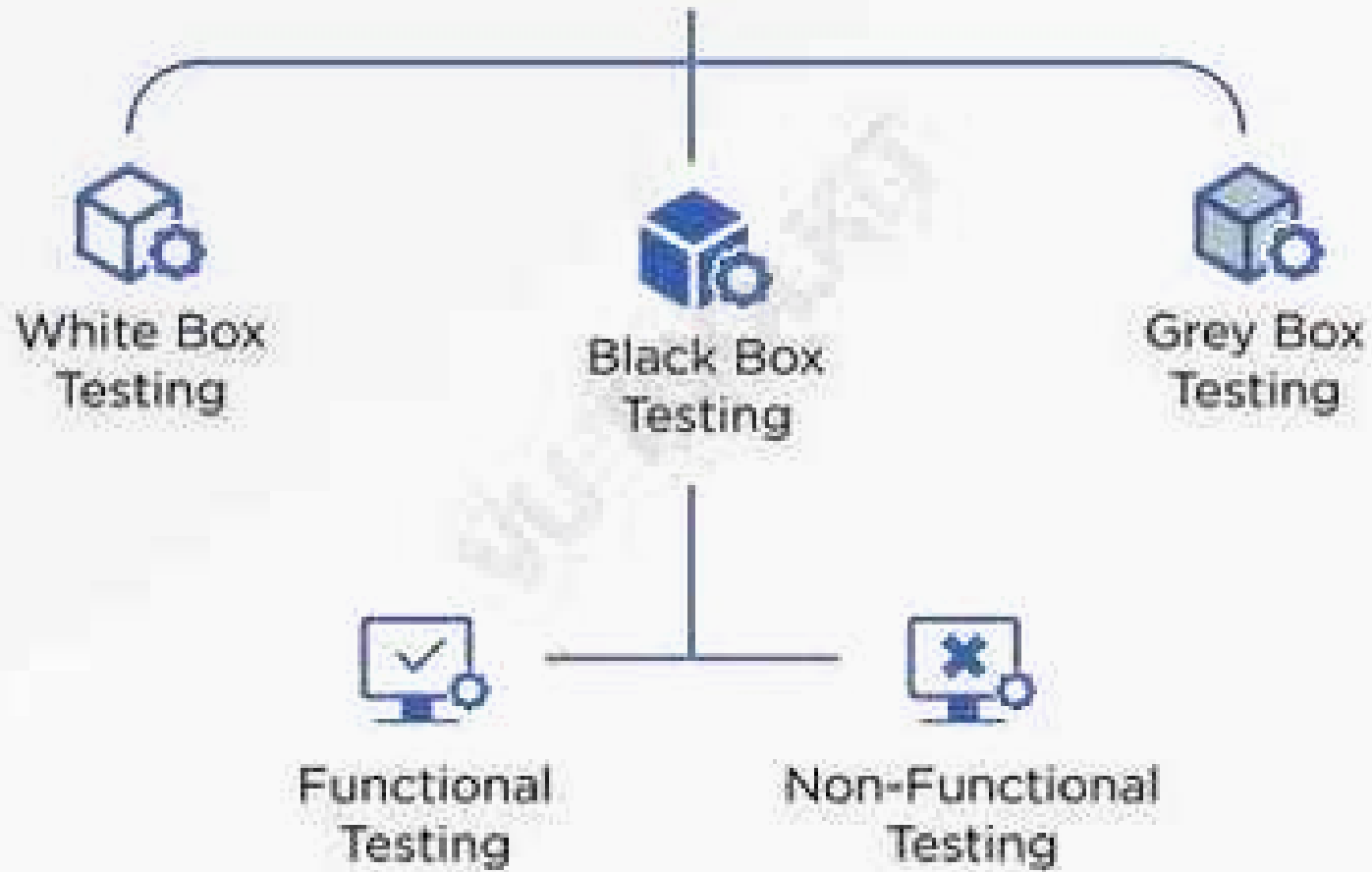
Coding

- The coding is the process of transforming the design of a system into a computer language format.



Testing

Approaches to Software Testing



Deployment



Maintenance

- Systems evolve after deployment.
- Continued improvements and changes to your systems as needed.



Need of well-defined Software Development

Following a well-defined methodology allows development teams to:

- Produce stable systems,
- Ensure customers are informed,
- Have a clear understanding of the task at hand,
- Offer better estimates,
- Identify potential pitfalls early on in the project.

Software Development Life Cycle Models

- A software development life cycle (SDLC) refers to the various stages involved in system development.
- It may be focused on software, hardware, or a combination of both.

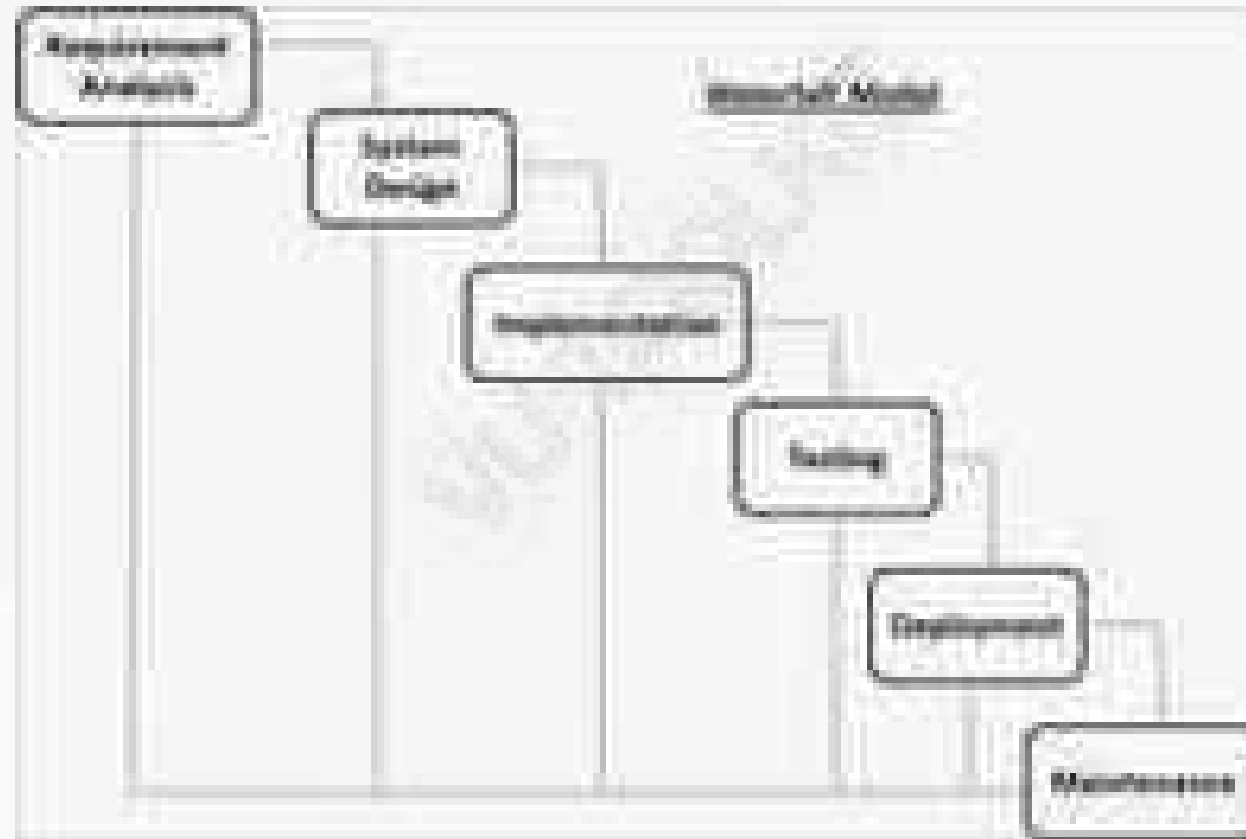
Most commonly used software development models are:

- Waterfall model
- Iterative Process models
- Agile Methodologies
- Evolutionary Prototyping

Waterfall Model

- Waterfall approach was first SDLC Model to be used widely in Software Engineering to ensure success of the project.
- In "The Waterfall" approach, the whole process of software development is divided into separate phases.
- In this Waterfall model, typically, the outcome of one phase acts as the input for the next phase sequentially.

Waterfall Model



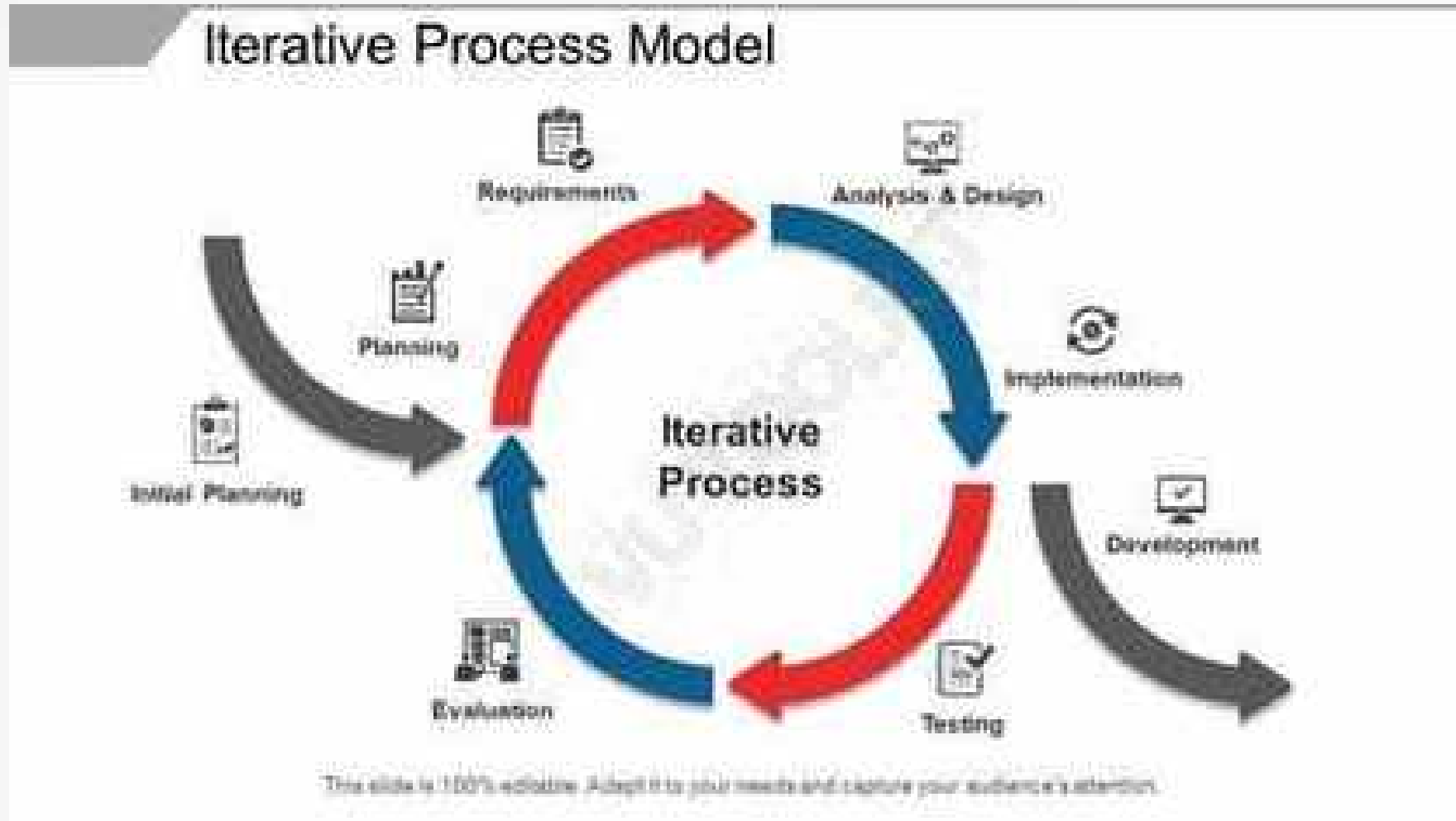
Waterfall Model advantages and disadvantages



Iterative Process models

- In this Model, you can start with some of the software specifications and develop the first version of the software.
- After the first version if there is a need to change the software, then a new version of the software is created with a new iteration.
- Every release of the Iterative Model finishes in an exact and fixed period that is called iteration.
- The Iterative Model allows the accessing earlier phases, in which the variations made respectively.
- The final output of the project renewed at the end of the Software Development Life Cycle (SDLC) process.

Iterative Process models



<https://www.slideteam.net/iterative-process-model.html>



Iterative Process models advantages and disadvantages

Iterative Model

- Advantages
 - Complexity broken down
 - Problems detected early
 - Allows evolution of requirements
 - Smaller development teams
- Disadvantages
 - Risk Analysis not formalized
 - Less parallelism



Iterative Process model - RUP

- Rational Unified Process-

- Document Driven approach
- An iterative and incremental approach
- Extensive planning,
- Codified Process,
- Heavy Documentation ,
- Big Design up Front

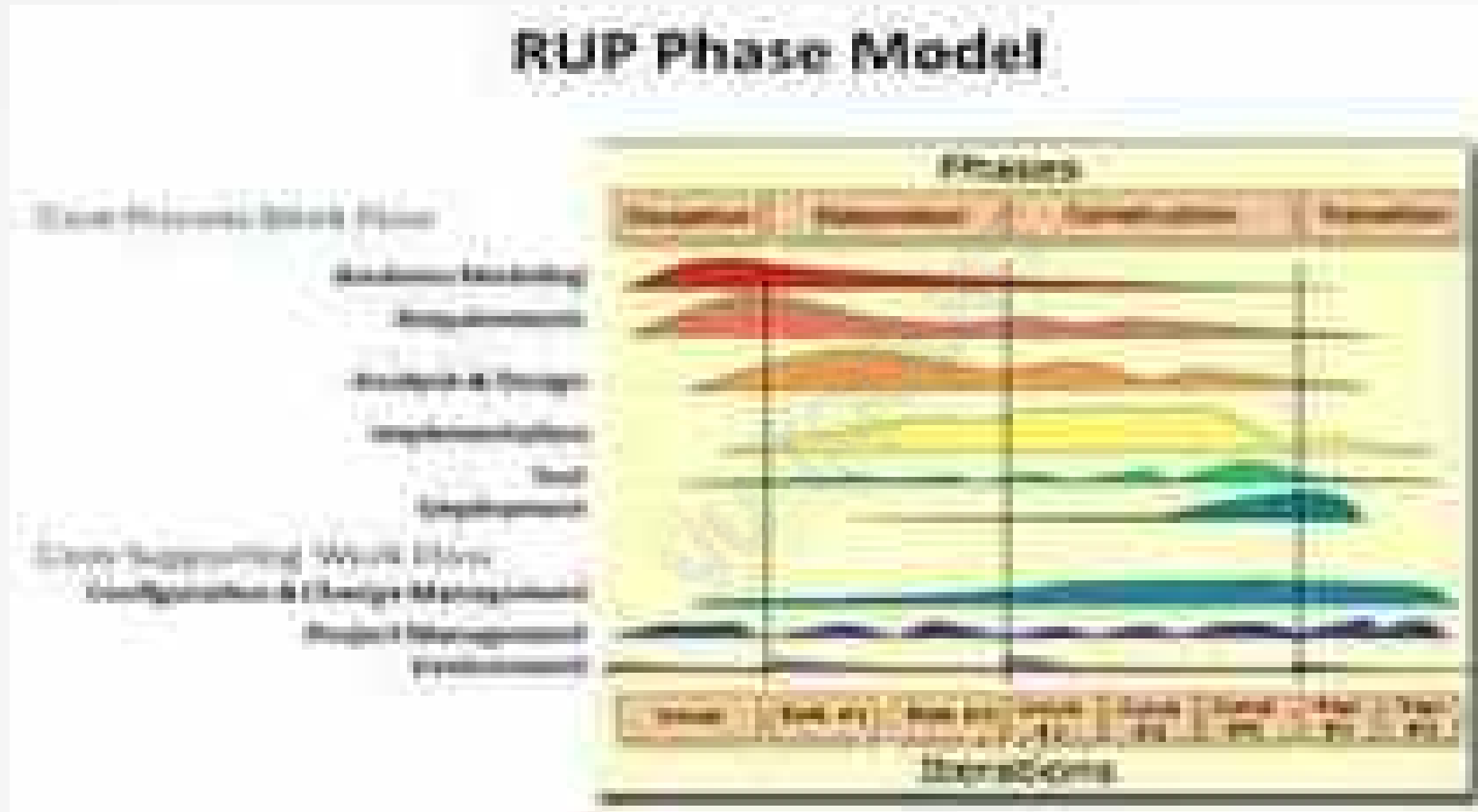
RUP Phases:

Inception, Elaboration, Construction and Transition

RUP Disciplines:

Main disciplines and Support disciplines

Iterative Process model - RUP



<https://www.mindcrafts.ch/2014/03/12/rup-rational-unified-process/>

RUP Strengths and Weaknesses

Strengths:

- Straightforward, methodical and structured nature, Predictability, stability and high assurance

Weaknesses

- Slow adaptation to rapidly changing business requirements
- A tendency to be over budget
- A tendency to be behind schedule
- Failed to provide dramatic progress in productivity, simplicity and reliability

Agile Methodologies

- The meaning of Agile is swift or versatile. "Agile process model" refers to a software development approach based on iterative development.
- Agile methods break tasks into smaller iterations, or parts do not directly involve long term planning.
- Each iteration is considered as a short time "frame" in the Agile process model, which typically lasts from one to four weeks.

Agile Manifesto

- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.

Agile Software Development Characteristics

- Modularity:
- Iterative:
- Adaptive:
- Incremental and Risk Proactive:
- People Oriented:
- Collaborative:

References

- <https://www.ibm.com/topics/software-development>
- <https://blog.bydrec.com/key-benefits-of-the-software-development-life-cycle>
- <https://www.javatpoint.com/software-engineering-iterative-model>
- https://www.teach-ict.com/as_a2_ict_new/ocr/A2_G063/331_systems_cycle/prototyping_RAD/miniweb/pg3.htm
- <https://www.geeksforgeeks.org/overview-of-plan-driven-development-pdd/>

References

- <https://slidemodel.com/requirements-gathering-project-management/>
- <https://www.educba.com/software-system-architecture/>
- <https://www.geeksforgeeks.org/introduction-of-software-design-process-set-2/>
- <https://www.javatpoint.com/software-engineering-coding>
- <https://www.leewayhertz.com/software-testing-process/>
- <https://www.mcgilly.com/services/deployment-and-maintenance/>

Software Construction & Development

Week-3

Agile Methodologies



Phases of Agile Model:

1. Requirements gathering:
2. Design the requirements:
3. Construction/ iteration:
4. Testing:
5. Deployment:
6. Feedback:

Agile Methods:

- Scrum
- Crystal
- Dynamic Software Development Method(DSDM)
- Feature Driven Development(FDD)
- Lean Software Development
- eXtreme Programming(XP)

When to use the Agile Model?

- When frequent changes are required.
- When a highly qualified and experienced team is available.
- When a customer is ready to have a meeting with a software team all the time.
- When project size is small.

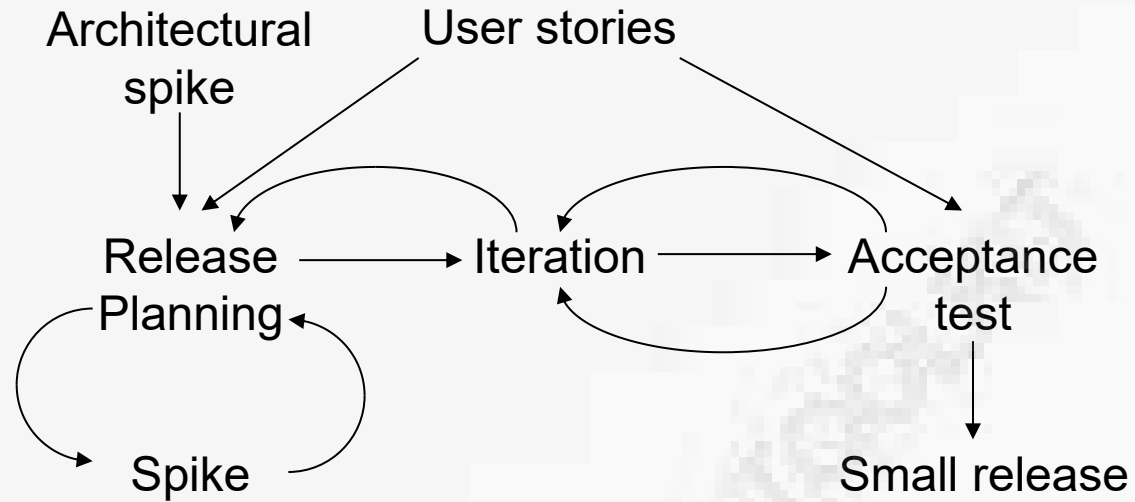
Advantage(Pros) of Agile Method:

- Frequent Delivery
- Face-to-Face Communication with clients.
- Efficient design and fulfils the business requirement.
- Anytime changes are acceptable.
- It reduces total development time.

Extreme Programming (XP) characteristics:

- Iterative and Incremental development,
- Customer collaboration,
- Frequent Delivery,
- Light and fast development,
- Light documentation

Extreme Programming Working:



XP Practices:

12 Practices

- The Planning Game:
- Small Releases:
- Metaphor:
- Simple Design:
- Tests:
- Refactoring:

XP Practices:

- Pair Programming:
- Collective Ownership:
- Continuous Integration:
- 40-hour Week:
- On-site customer:
- Coding Standard:

XP Strengths:

- More customer-centric approach
- Welcome change in requirements
- More satisfactory relationship with the client
- Dramatic progress in productivity, simplicity and reliability
- Best engineering practices
- Tendency to be within budget and as per schedule
- Built highest priority first .
- Lower software bugs rates

XP Weaknesses:

- Not suitable for medium, complex and large scale projects.
- XP is not suitable in case of stable requirements.
- XP mainly focuses on people rather than processes
- XP doesn't focus to model a well defined mechanism for business modeling.
- Not a well-known approach to handle non-functional requirements.
- XP doesn't produce good quality results with new technology.
XP doesn't offer to capture user requirements through different elicitation techniques.

Scrum Development Methodology:

- Short development cycle
- Higher customer satisfaction
- Low bug rates
- Quick adaptation to rapidly changing requirements
- Highest Priority Work
- Constant Feedback
- Control over Cost and Schedule

Scrum basic Concepts:

- Scrum is another popular agile technique
- **Sprint:**
- The development cycle in Scrum is called Sprint which lasts for approximately 30 days. In Scrum there are two backlogs;
Product Backlog: “All work to be performed in the foreseeable future, both well defined and requiring further definition”
- **Sprint Backlog:** “That work that is well-enough defined that it can be worked on with relatively little change over a period of 30 days or less and will result in a tangible, incremental deliverable”.

Scrum Roles and Responsibilities:

Product Owner

- Responsibility is to define business value, requirements of project and Prioritization.

• Scrum Master

- Scrum Master is the Project Manager of the project. Scrum Master has to lead the project and make sure the software should meet the requirements set by the product owner

• Scrum Development Team:

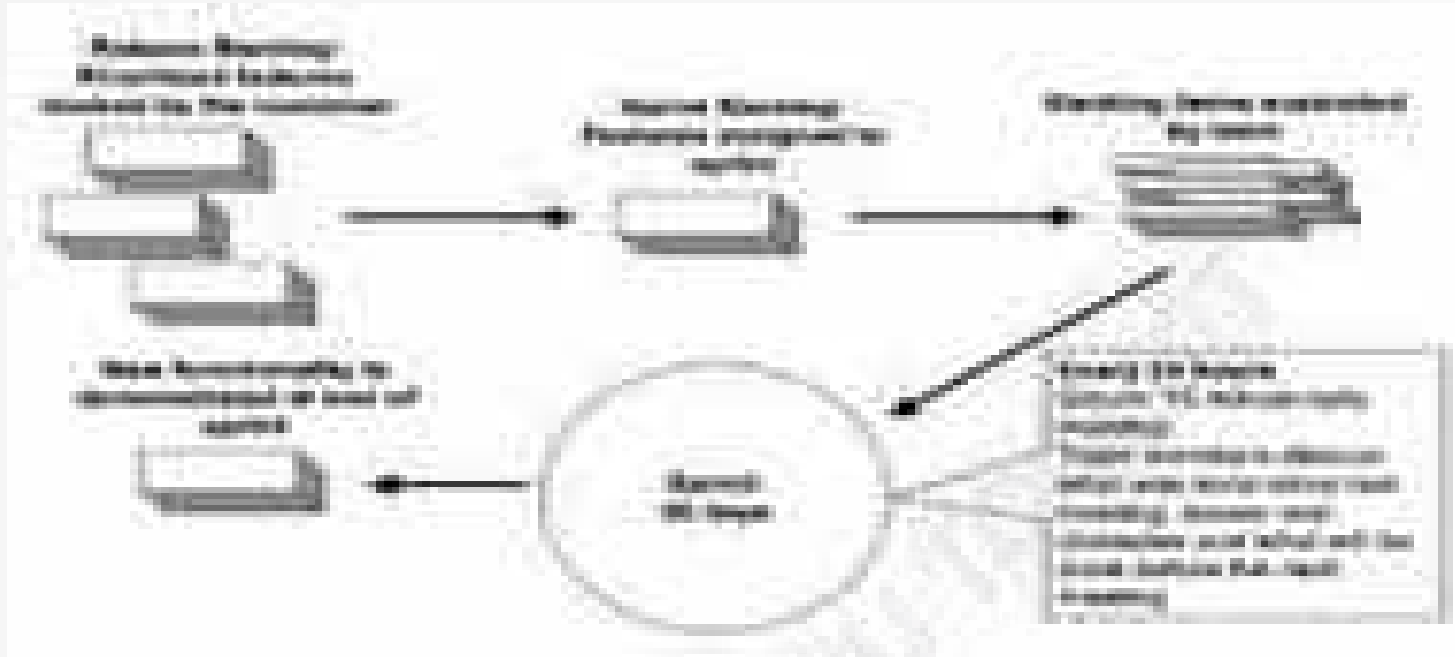
- The responsibility of the Team is to meet the goal as well as the deadline. The team is dedicated to one project only.

lifecycle of a Sprint (iteration):

- Pre-Sprint Planning
- The Sprint Planning Meeting
- The Sprint Review Meeting
- The Sprint Retrospective
- A quality assurance practice
- The Daily Scrum Meeting

Scrum Working:

-



David Cohen. et Al. (2003)

Scrum Roles

Evolutionary Prototyping

- The idea behind this is that an initial prototype is presented to the user.
- They provide feedback and suggestions for improvements.
- These are actioned by the developer who then presents a more refined prototype.
- The user once more provides feedback. The process is repeated.
- So at each stage the prototype 'evolves' towards the final system.
- Hence the term 'evolutionary prototyping'.

Evolutionary Prototyping



Risk-with Plan Driven Models

- Disadvantages of Plan Driven Development:
 - Can not lodge changes any time.
 - Longer distance in each repetition or increase.
 - The shoulder that, future variations will not occur.
 - Lack of user participation through the life cycle of the creation.
- Expensive for the dynamic expansion environment.

Conventional Vs Agile

	Software Process Model	Advantages	Disadvantages
Plan Driven	<ul style="list-style-type: none"> Waterfall Incremental Development Roadmap Development Agile Development Prototype Model Rapid Application Development 	<ul style="list-style-type: none"> Good for large systems and teams. Handles highly stable or complex environments. Appropriate for stable development environments. Requires experienced personnel at the beginning. Requires a formal change control system and roles. 	<ul style="list-style-type: none"> Longer length of each iteration or milestone. Changes or requirements changes are slow. Lack of user involvement throughout the life cycle of the project. Only for the dynamic development environments. Assumes that future changes will not occur.
Agile	<ul style="list-style-type: none"> Scrum Model Extreme Programming (XP) Dynamic Systems Development Method Kanban Feature Driven Development 	<ul style="list-style-type: none"> Good for small to medium systems and teams. Can accommodate changes at any time. Effective for the dynamic development environments. Requires expert agile personnel throughout the life cycle. Requires a formal change control system and roles. 	<ul style="list-style-type: none"> Not suitable for large systems (many SRSs). Shorter length of each iteration. Can accommodate changes at any time. Only for the stable development environments. Assumes that frequent future changes will occur.

https://www.researchgate.net/publication/329357135_ScrumFall_A_Hybrid_Software_Process_Model/figures?lo=1

References

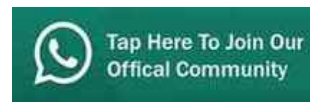
- <https://www.ibm.com/topics/software-development>
- <https://blog.bydrec.com/key-benefits-of-the-software-development-life-cycle>
- <https://www.javatpoint.com/software-engineering-iterative-model>
- https://www.teach-ict.com/as_a2_ict_new/ocr/A2_G063/331_systems_cycle/prototyping_RAD/miniweb/pg3.htm
- <https://www.geeksforgeeks.org/overview-of-plan-driven-development-pdd/>

References

- <https://slidemodel.com/requirements-gathering-project-management/>
- <https://www.educba.com/software-system-architecture/>
- <https://www.geeksforgeeks.org/introduction-of-software-design-process-set-2/>
- <https://www.javatpoint.com/software-engineering-coding>
- <https://www.leewayhertz.com/software-testing-process/>
- <https://www.mcgilly.com/services/deployment-and-maintenance/>

User interface design

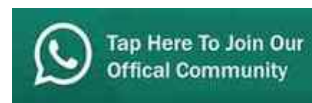
λ Designing effective interfaces
for software systems



Software GUI Construction

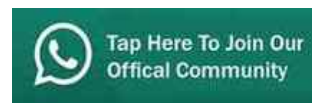
To suggest some general design principles for user interface design

- λ To explain different interaction styles
- λ To introduce styles of information presentation
- λ To describe the user support which should be built-in to user interfaces
- λ To introduce usability attributes and system approaches to system evaluation



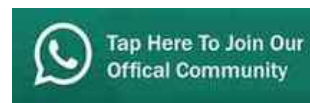
The user interface

- λ System users often judge a system by its interface rather than its functionality
- λ A poorly designed interface can cause a user to make catastrophic errors
- λ Poor user interface design is the reason why so many software systems are never used



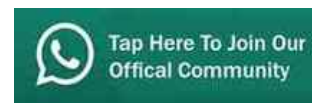
Graphical user interfaces

- λ Most users of business systems interact with these systems through graphical interfaces although, in some cases, legacy text-based interfaces are still used



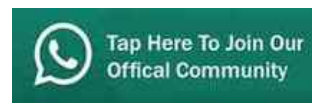
GUI characteristics

Characteristic	Description
Windows	Multiple windows allow different information to be displayed simultaneously on the user's screen.
Icons	Icons different types of information. On some systems, icons represent files; on others, icons represent processes.
Menus	Commands are selected from a menu rather than typed in a command language.
Pointing	A pointing device such as a mouse is used for selecting choices from a menu or indicating items of interest in a window.
Graphics	Graphical elements can be mixed with text on the same display.

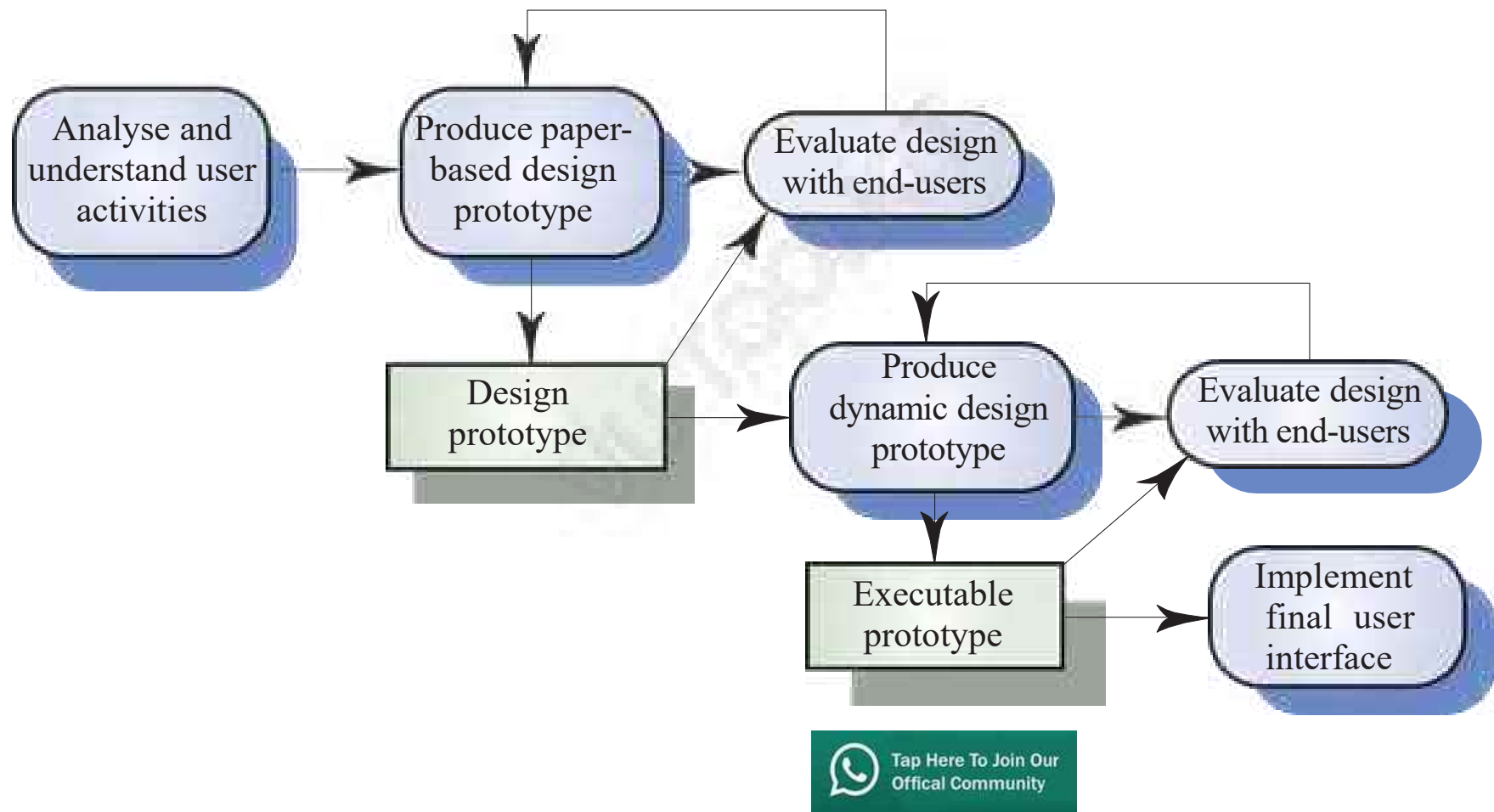


GUI advantages

- λ They are easy to learn and use.
 - Users without experience can learn to use the system quickly.
- λ The user may switch quickly from one task to another and can interact with several different applications.
 - Information remains visible in its own window when attention is switched.
- λ Fast, full-screen interaction is possible with immediate access to anywhere on the screen

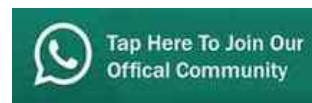


User interface design process



UI design principles

- λ UI design must take account of the needs, experience and capabilities of the system users
- λ Designers should be aware of people's physical and mental limitations (e.g. limited short-term memory) and should recognise that people make mistakes
- λ UI design principles underlie interface designs although not all principles are applicable to all designs



User interface design principles

Principle	Description
User familiarity	The interface should use terms and concepts which are drawn from the experience of the people who will make most use of the system.
Consistency	The interface should be consistent in that, wherever possible, comparable operations should be activated in the same way.
Minimal surprise	Users should never be surprised by the behaviour of a system.
Recoverability	The interface should include mechanisms to allow users to recover from errors.
User guidance	The interface should provide meaningful feedback when errors occur and provide context-sensitive user help facilities.
User diversity	The interface should provide appropriate interaction facilities for different types of system user.



Design principles

λ User familiarity

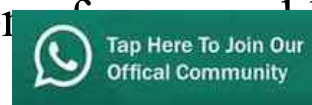
- The interface should be based on user-oriented terms and concepts rather than computer concepts. For example, an office system should use concepts such as letters, documents, folders etc. rather than directories, file identifiers, etc.

λ Consistency

- The system should display an appropriate level of consistency. Commands and menus should have the same format, command punctuation should be similar, etc.

λ Minimal surprise

- If a command operates in a known way, the user should be able to predict the operation of similar commands



Design principles

λ Recoverability

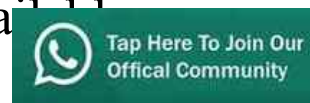
- The system should provide some resilience to user errors and allow the user to recover from errors. This might include an undo facility, confirmation of destructive actions, 'soft' deletes, etc.

λ User guidance

- Some user guidance such as help systems, on-line manuals, etc. should be supplied

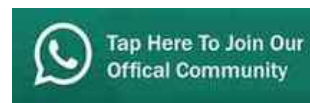
λ User diversity

- Interaction facilities for different types of user should be supported. For example, some users have seeing difficulties and so larger text should be available.



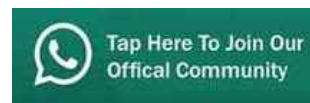
User-system interaction

- λ Two problems must be addressed in interactive systems design
 - How should information from the user be provided to the computer system?
 - How should information from the computer system be presented to the user?
- λ User interaction and information presentation may be integrated through a coherent framework such as a user interface metaphor

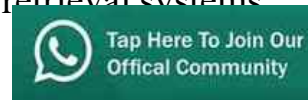


Interaction styles

- λ Direct manipulation
- λ Menu selection
- λ Form fill-in
- λ Command language
- λ Natural language



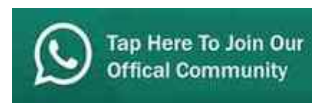
Interaction style	Main advantages	Main disadvantages	Application examples
Direct manipulation	Fast and intuitive interaction Easy to learn	May be hard to implement Only suitable where there is a visual metaphor for tasks and objects	Video games CAD systems
Menu selection	Avoids user error Little typing required	Slow for experienced users Can become complex if many menu options	Most general-purpose systems
Form fill-in	Simple data entry Easy to learn	Takes up a lot of screen space	Stock control, Personal loan processing
Command language	Powerful and flexible	Hard to learn Poor error management	Operating systems, Library information retrieval systems
Natural language	Accessible to casual users Easily extended	Requires more typing Natural language understanding systems are unreliable	Timetable systems WWW information retrieval systems



Advantages and disadvantages

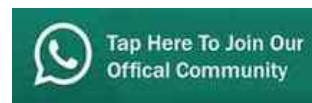
Direct manipulation advantages

- λ Users feel in control of the computer and are less likely to be intimidated by it
- λ User learning time is relatively short
- λ Users get immediate feedback on their actions so mistakes can be quickly detected and corrected



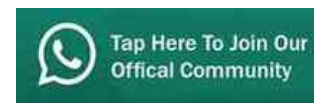
Direct manipulation problems

- λ The derivation of an appropriate information space model can be very difficult
- λ Given that users have a large information space, what facilities for navigating around that space should be provided?
- λ Direct manipulation interfaces can be complex to program and make heavy demands on the computer system



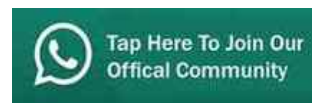
Control panel interface

Title	JSD. example	☐	Grid	Busy
Method	JSD			
Type	Network	Units	cm ▶	QUIT
Selection	Process	Reduce	Full ▶	PRINT
NODE LINKS FONT LABEL EDIT				



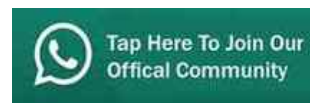
Menu systems

- λ Users make a selection from a list of possibilities presented to them by the system
- λ The selection may be made by pointing and clicking with a mouse, using cursor keys or by typing the name of the selection
- λ May make use of simple-to-use terminals such as touchscreens



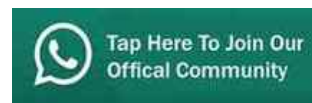
Advantages of menu systems

- λ Users need not remember command names as they are always presented with a list of valid commands 
- λ Typing effort is minimal
- λ User errors are trapped by the interface
- λ Context-dependent help can be provided. The user's context is indicated by the current menu selection



Problems with menu systems

- λ Actions which involve logical conjunction (and) or disjunction (or) are awkward to represent
- λ Menu systems are best suited to presenting a small number of choices. If there are many choices, some menu structuring facility must be used
- λ Experienced users find menus slower than command language



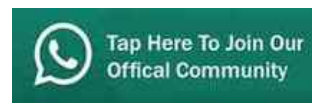
Form-based interface

NEW BOOK			
Title	<input type="text"/>	ISBN	<input type="text"/>
Author	<input type="text"/>	Price	<input type="text"/>
Publisher	<input type="text"/>	Publication date	<input type="text"/>
Edition	<input type="text"/>	Number of copies	<input type="text"/>
Classification	<input type="text"/>	Loan status	<input type="text"/>
Date of purchase	<input type="text"/>	Order status	<input type="text"/>

 Tap Here To Join Our Official Community

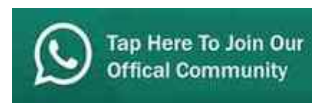
Command interfaces

- λ User types commands to give instructions to the system e.g. UNIX
- λ May be implemented using cheap terminals.
- λ Easy to process using compiler techniques
- λ Commands of arbitrary complexity can be created by command combination
- λ Concise interfaces requiring minimal typing can be created



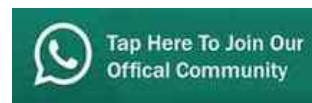
Problems with command interfaces

- λ Users have to learn and remember a command language. Command interfaces are therefore unsuitable for occasional users
- λ Users make errors in command. An error detection and recovery system is required
- λ System interaction is through a keyboard so typing ability is required



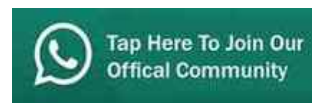
Command languages

- λ Often preferred by experienced users because they allow for faster interaction with the system
- λ Not suitable for casual or inexperienced users
- λ May be provided as an alternative to menu commands (keyboard shortcuts). In some cases, a command language interface and a menu-based interface are supported at the same time

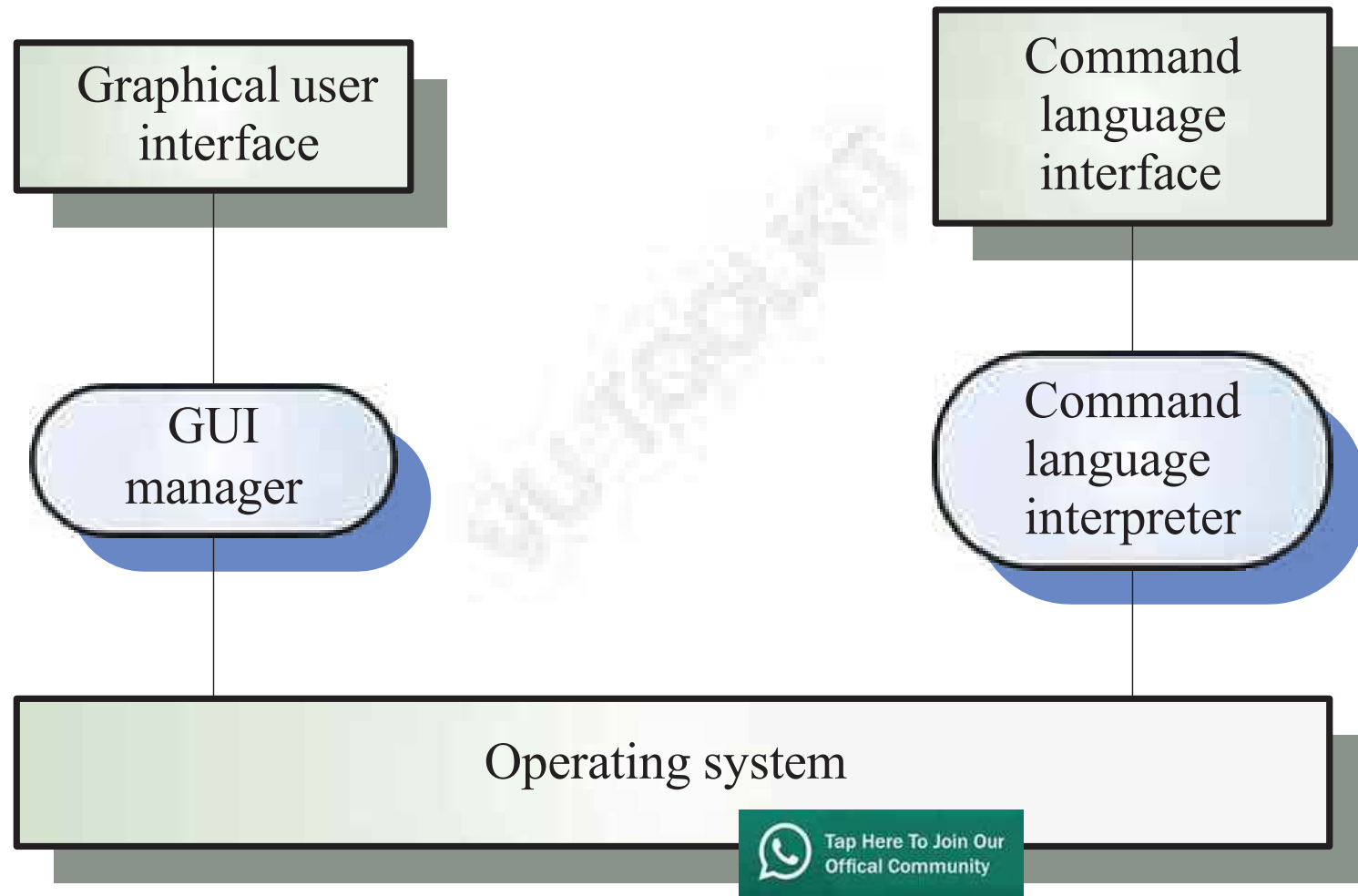


Natural language interfaces

- λ The user types a command in a natural language. Generally, the vocabulary is limited and these systems are confined to specific application domains (e.g. timetable enquiries)
- λ NL processing technology is now good enough to make these interfaces effective for casual users but experienced users find that they require too much typing

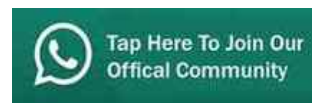


Multiple user interfaces

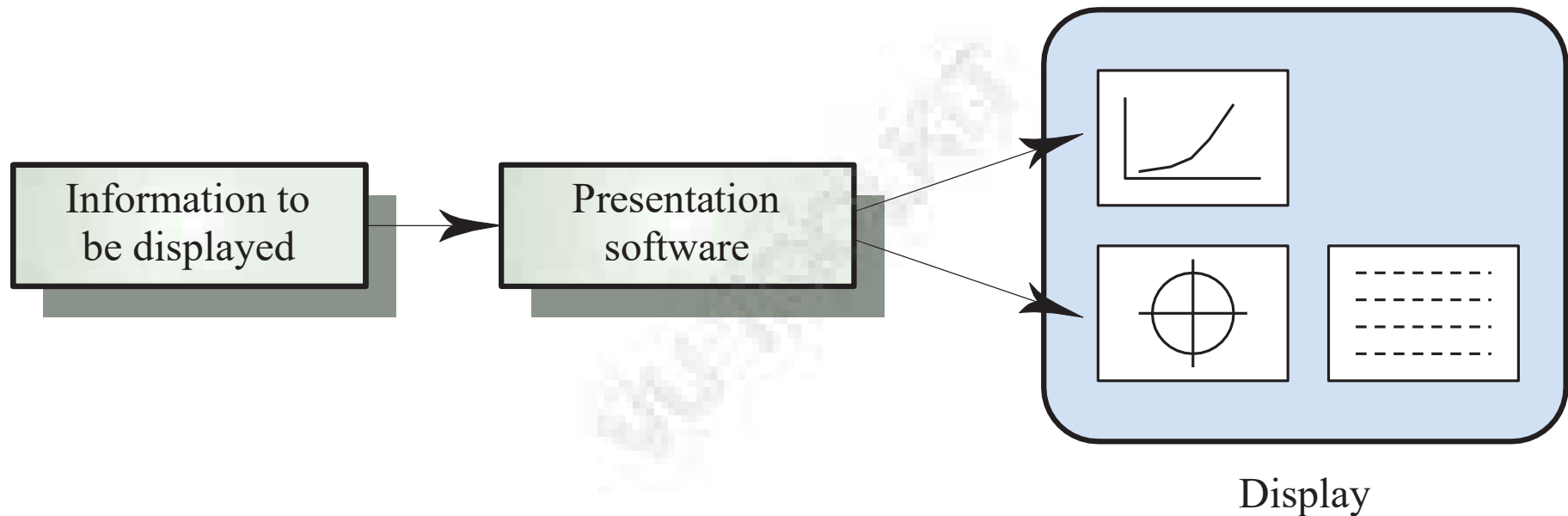
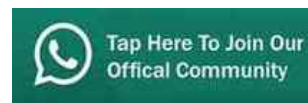


Information presentation

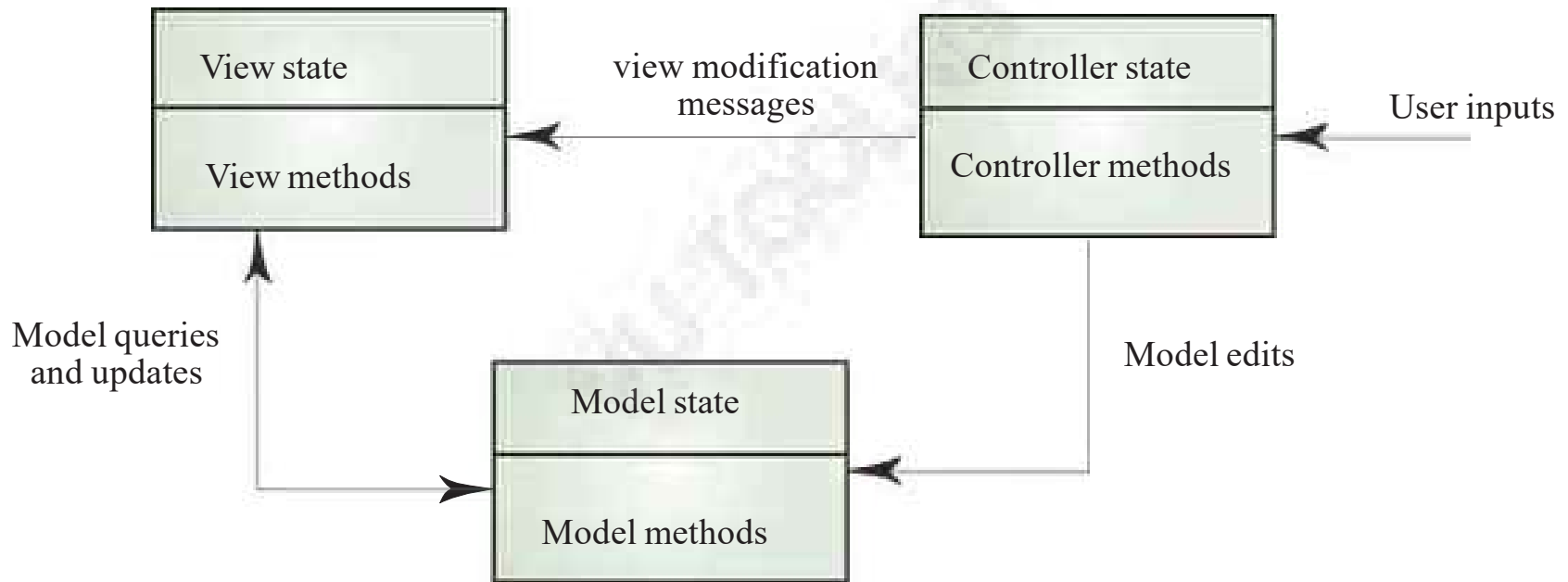
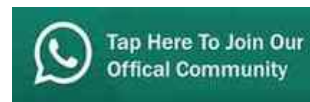
- λ Information presentation is concerned with presenting system information to system users
- λ The information may be presented directly (e.g. text in a word processor) or may be transformed in some way for presentation (e.g. in some graphical form)
- λ The Model-View-Controller approach is a way of supporting multiple presentations of data



Information presentation



Model-view-controller



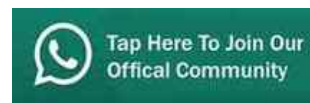
Information presentation

λ Static information

- Initialised at the beginning of a session. It does not change during the session
- May be either numeric or textual

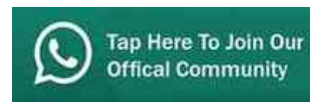
λ Dynamic information

- Changes during a session and the changes must be communicated to the system user
- May be either numeric or textual



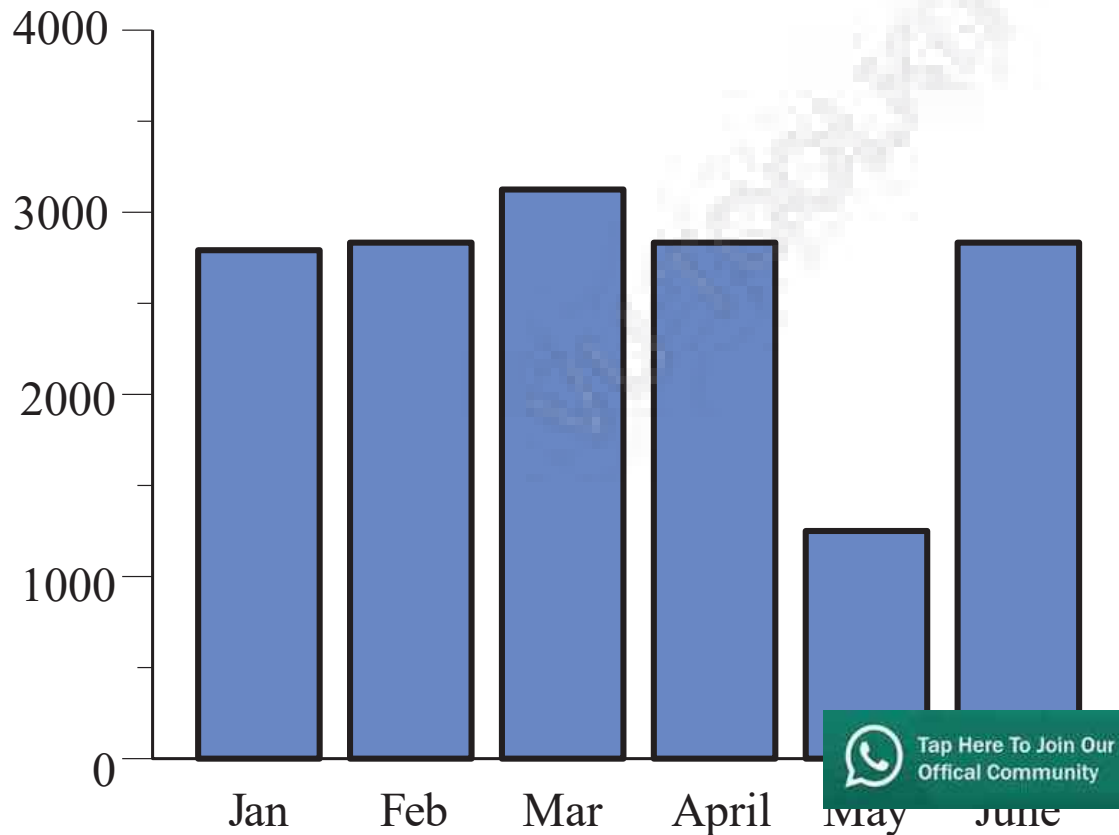
Information display factors

- λ Is the user interested in precise information or data relationships?
- λ How quickly do information values change? Must the change be indicated immediately?
- λ Must the user take some action in response to a change?
- λ Is there a direct manipulation interface?
- λ Is the information textual or numeric? Are relative values important?



Alternative information presentations

Jan	Feb	Mar	April	May	June
2842	2851	3164	2789	1273	2835



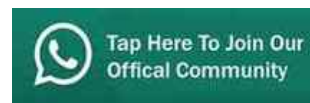
Analogue vs. digital presentation

λ Digital presentation

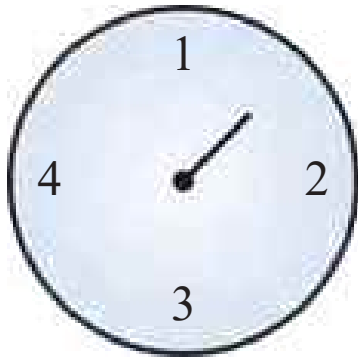
- Compact - takes up little screen space
- Precise values can be communicated

λ Analogue presentation

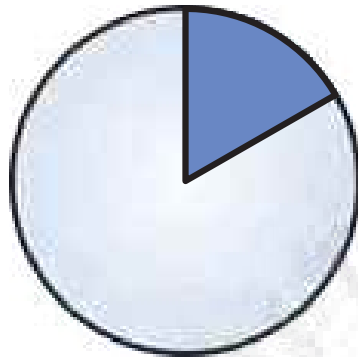
- Easier to get an 'at a glance' impression of a value
- Possible to show relative values
- Easier to see exceptional data values



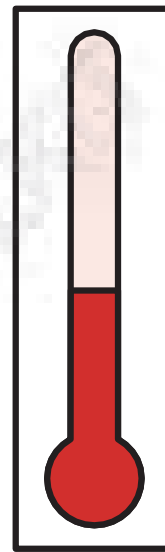
Dynamic information display



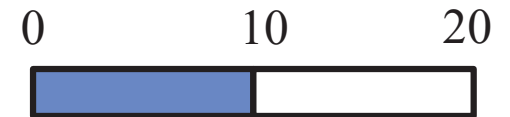
Dial with needle



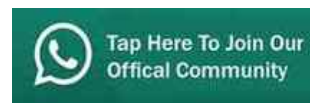
Pie chart



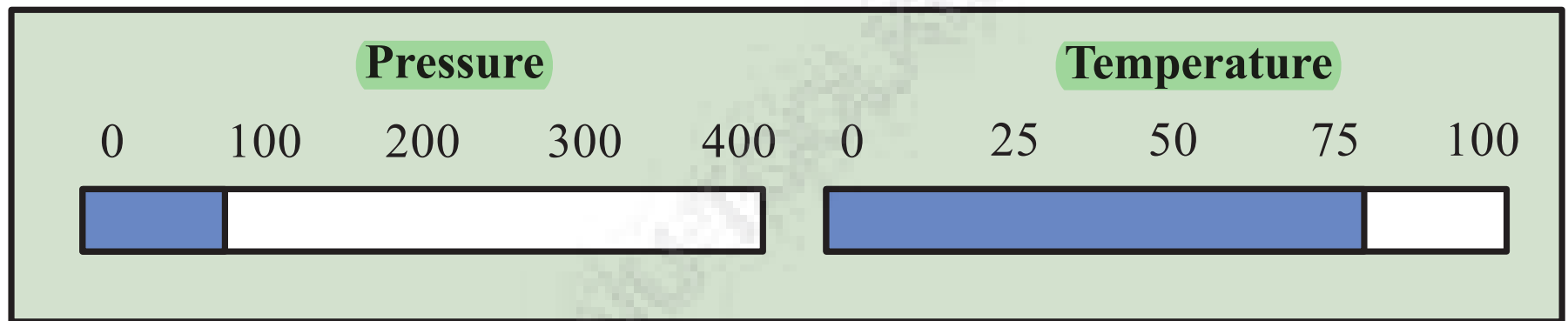
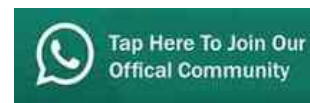
Thermometer



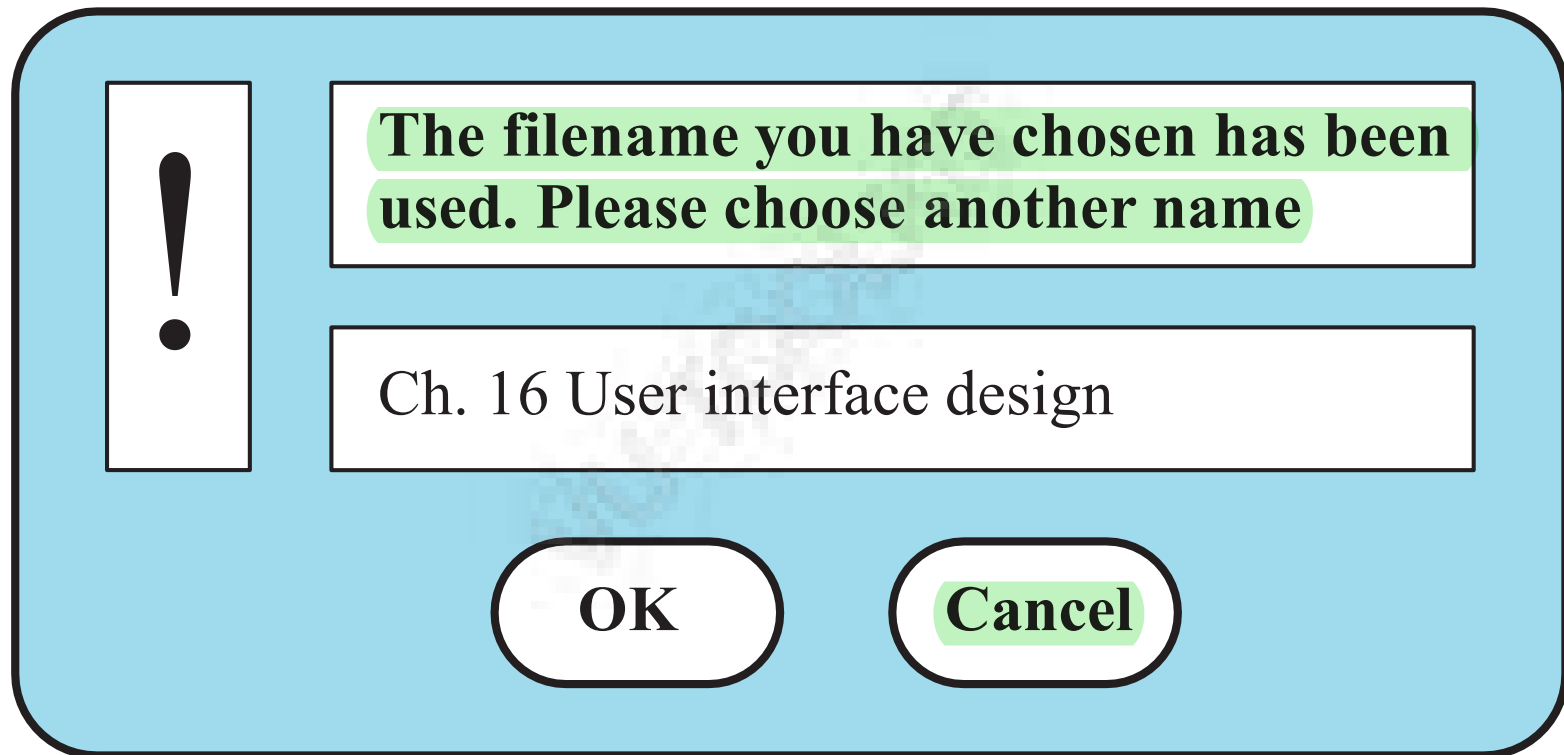
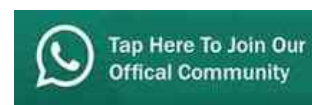
Horizontal bar



Displaying relative values

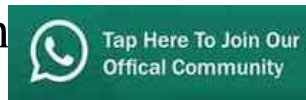


Textual highlighting



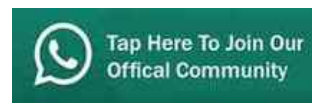
Data visualisation

- λ Concerned with techniques for displaying large amounts of information
- λ Visualisation can reveal relationships between entities and trends in the data
- λ Possible data visualisations are:
 - Weather information collected from a number of sources
 - The state of a telephone network as a linked set of nodes
 - Chemical plant visualised by showing pressures and temperatures in a linked set of tanks and pipes
 - A model of a molecule displayed in 3 dimensions
 - Web pages displayed as a h



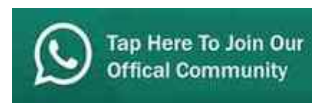
Colour displays

- λ Colour adds an extra dimension to an interface and can help the user understand complex information structures
- λ Can be used to highlight exceptional events
- λ Common mistakes in the use of colour in interface design include:
 - The use of colour to communicate meaning
 - Over-use of colour in the display



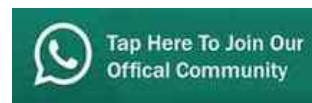
Colour use guidelines

- λ Don't use too many colours
- λ Use colour coding to support use tasks
- λ Allow users to control colour coding
- λ Design for monochrome then add colour
- λ Use colour coding consistently
- λ Avoid colour pairings which clash
- λ Use colour change to show status change
- λ Be aware that colour displays are usually lower resolution

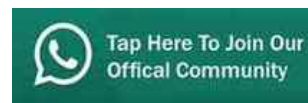


User support

- λ User guidance covers all system facilities to support users including on-line help, error messages, manuals etc.
- λ The user guidance system should be integrated with the user interface to help users when they need information about the system or when they make some kind of error
- λ The help and message system should, if possible, be integrated



What is usability



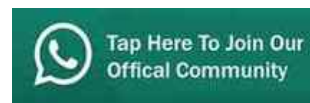
Usability

Definition :

ISO 9241-210, defines usability as “extent to which a system, product or service can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use”.

Usability Perspectives

- (i) A **product-oriented perspective** emphasizes that usability can be measured with regard to the ergonomic characteristics of a product.
- (ii) A **user-oriented perspective**, usability is represented by means of the user's mental effort in product usage and his/her attitude towards the product.
- (iii) A **performance-oriented perspective**, usability is described in terms of the interaction of the user with the product,
- (iv) A **context-oriented perspective** emphasizes that usability is dependent on the user group that is studied, the tasks those users are performing, and the environment in which the tasks are completed.

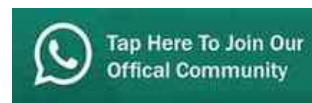


Usability Engineering

Usability is all about how users interact with technology

Usability engineering requires a firm knowledge of computer science and psychology and approaches product development based on customer feedback.

A usability engineer works hand-in-hand with customers, working to develop a better understanding of the functionality

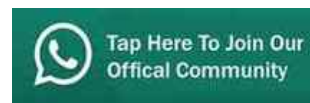


Why isn't UE done Regularly?

Developers believe

- it takes too long
- is too expensive
- is not critical to development
- they can afford to learn about user problems during “live” use

Also developers may not know how to do it



Benefits of UE to an Organization

Reduce training costs

Reduce development costs

Identify and fix problems early

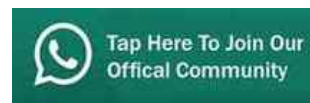
Reduce support costs; minimize need for

support personnel/help desks

fixes, maintenance, upgrades

Enhance organization's reputation - positive "word-of-mouth" trade

Larger numbers of "hit" and "return visit" rates

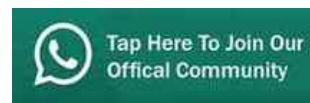


Benefits of UE to the User

Less time to complete work

Greater success with tasks

Increased user satisfaction



Typical Web-site Usability Problems

Navigation

- Knowing where you are

- Finding what you want

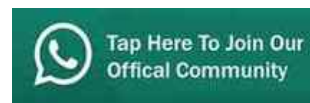
Structure of web site

Layout

- Needs sufficient white space

- Use of large graphics

“The Google problem”



User Centered Design

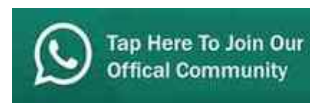
UCD is a dialog between the customer and the designer

Rules of thumb:

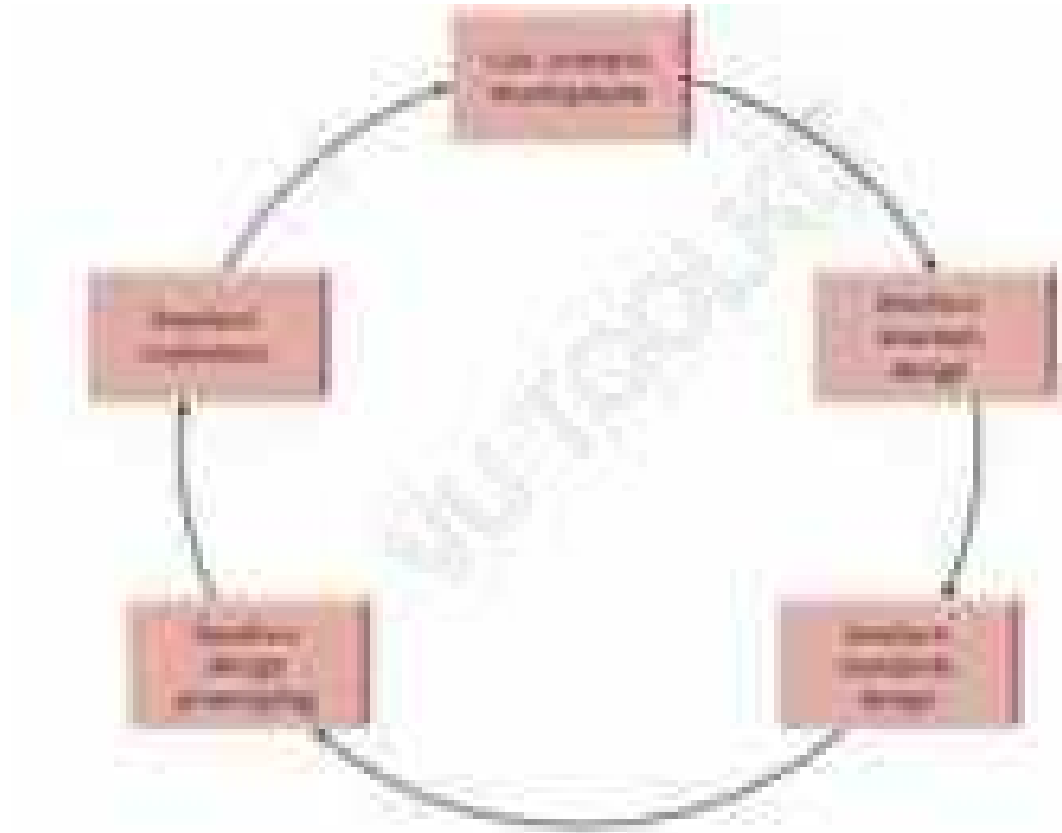
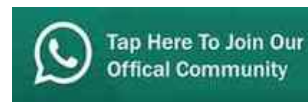
- Get to know and understand the users.

- Build an application, applying usability principles.

- Test designs by observing users in a real work setting (environment and work load).



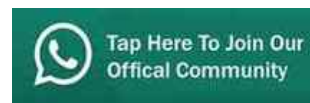
Usability design Process



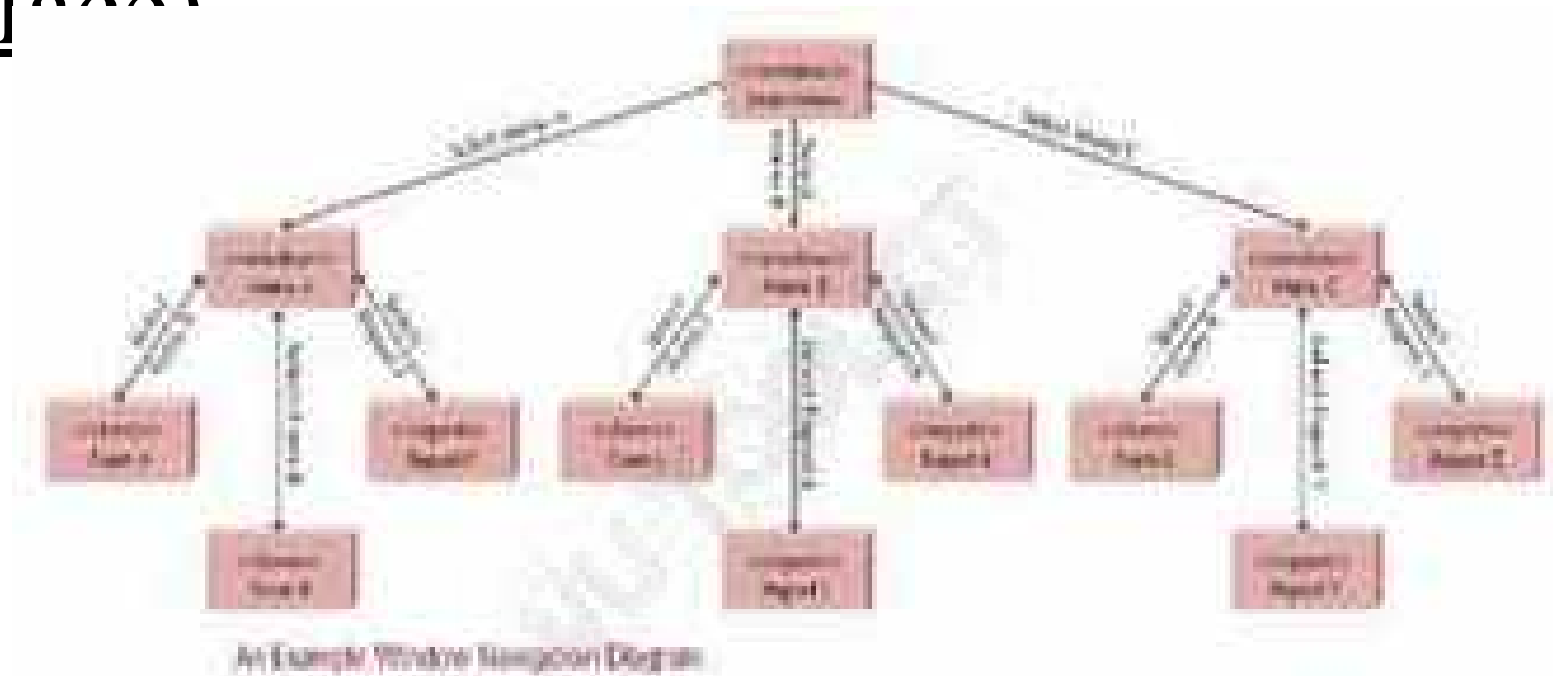
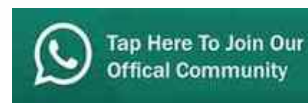
Use scenario Development

You already know about use cases

Use scenarios are just a single thread through a use case – much simpler!



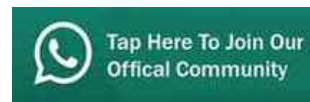
UI Structure (architecture diagram of interface)



Interface Standard Design (one video to another..what standard must follow)

UI design principles: key rules and guidelines

1. Make it easy to explore and use
2. Give users control of the interface
3. Create a layout that works efficiently
4. Offer a consistent interface
5. Use real-world metaphors in your UI
6. Know your way around design materials
7. Start with black-and-white UI design, then build on it
8. Feedback and context is important
9. Fine-tune your wireframing game
10. Get familiar with user testing and the world of usability



Interface Design Prototyping (develop storyboard on paper)

Storyboard

HTML Prototype

Language Prototype

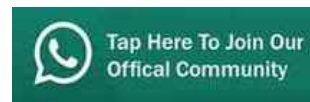
Interface Evaluation

Heuristic

Walkthrough

Interactive

Formal Usability Testing



Usability Principles

Principle	Description
Consistency	The interface should be a series of small, consistent, and predictable components. It is different from having a single, large, and complex component, which is not predictable and is not consistent.
Standardization	Users should be able to predict the behavior of the interface by using standard components.
Flexibility	Flexibility should be provided for users to learn through trial and error. It should be able to handle errors and provide feedback. It should be able to handle errors and provide feedback. It should be able to handle errors and provide feedback.
Local Help	Although users do not want to be interrupted, they do want to be able to get help. Local help should be provided for users to get help when they need it.
Consistency	Consistency is important for users to learn the interface. It should be consistent in terms of layout, color, and behavior. It should be consistent in terms of layout, color, and behavior. It should be consistent in terms of layout, color, and behavior.
Minimalist Interface	The interface should be simple and easy to use. It should not be cluttered with unnecessary elements. It should be simple and easy to use. It should not be cluttered with unnecessary elements. It should be simple and easy to use.



Tap Here To Join Our
Official Community

GUI Design is Multi Disciplinary

A team includes

Analyst

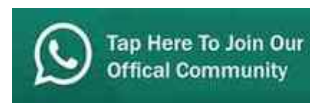
Designer

Technology expert

Graphic artist

Social and behavioral scientist

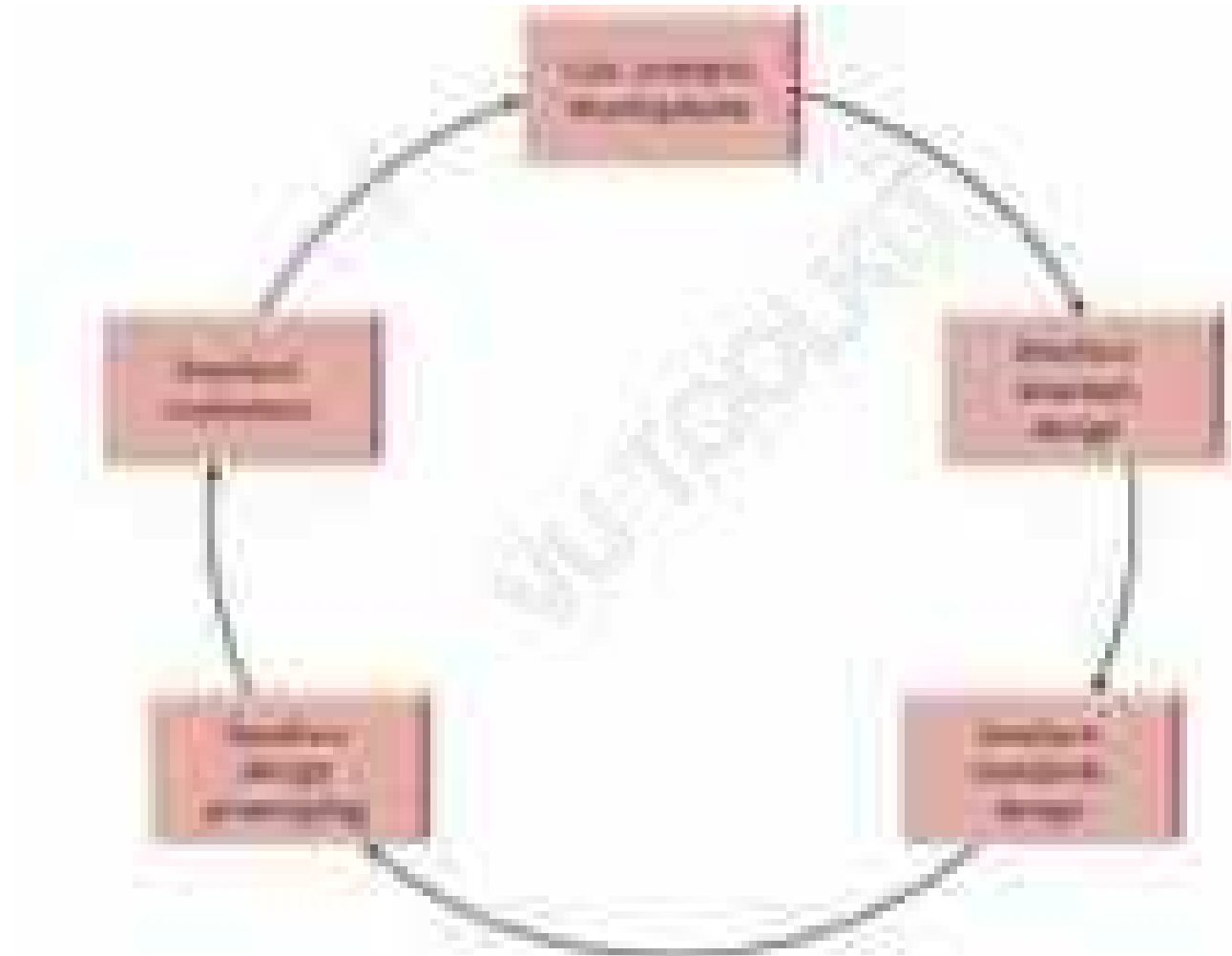
Programmer



Software Construction and Development

Week-7

Usability design Process



Usability evaluation of Interfaces

- Definition

- consists of methodologies for measuring the usability aspects of a system's user interface (UI) and identifying specific problems

- Usability Evaluation Methods

- testing, inspection, and inquiry

- Approaches to evaluate usability

- Summative

- Real users perform the real tasks on some defined applications and empirically measure the usability in terms of effectiveness, efficiency and user satisfaction.

- Formative

- Formative approach employs the experts for UE using heuristics or guidelines

Heuristics for Interface Usability Evaluation

- An inspection based usability evaluation method
- Consists a systematic evaluation in which several experts—usually 3 to 5—analyze the interface of an interactive system and make observations based on a set of principles known as usability heuristics.
- The goal of heuristic evaluation is to discover problems related to the usability in the interface design
- Protocol of a typical heuristic evaluation involves that each evaluator spends time inspecting the system interface in an independent manner, making note of every usability problem found. Once all evaluators have complete the evaluation, their individual findings are joined in a unique common list of usability problems associated to those heuristics that were violated. The evaluators must rate the severity of each problem, according to a scale of 0 (lowest severity) to 4 (highest severity)
- Nielsen proposed 10 general usability heuristics for the design and evaluation of interfaces, Known as “Nielsen’s heuristics”.

Design and Evaluation Heuristics for Smartphone : Case Study

- Smartphone Features with associated Usability Limitations:



Smartphone Features with associated Usability Limitations:

Small screen size	Small screen size may cause congestion of information on display area and the user can mistakenly click the incorrect option due to substantial information. On the other hand, if the mobile UI contain small amount of information, then, there are chances that the end user could not find some of the core information.
Limited input/output capabilities	Adapted gestures for interaction may change the value of mobile usability. Similarly, simulated keyboard also causes usability issues. Users concurrently spend the cognitive workload between the content and the keypad region.
Limited computational resources	User demands that smartphone should complete the task without any mistake and behave more likely a desktop.
Limited power,	More interface brightness, internet connectivity and working of multiple sensors requires more consumption of battery power. End user suffers and compromises in such cases.
Types of tasks	Mobile devices provide limited capability in it
Display resolution,	Comparatively, lower resolution of mobile resulting in lesser quality images.
Mobile Physical buttons	Small mobile buttons and labels can reduce the task effectiveness and efficiency.

Smartphone Features with associated Usability Limitations

Costly bandwidth and limited connectivity	The problem of slow and poor internet connection can reduce the tasks efficiency.
Wide heterogeneity	Users of mobile systems continuously adapt to new forms. They need to learn different actions, letters, gestures and others.
Mobility	User while moving around may distract due to different context of use including bad light, noise and others. These constraints effects the application usability.
kind of interaction	Multitouch gesture is also not defined in a well-organised way.
Interruption	Notifications and requests may reduce the task efficiency and user's comfort level.
Intimacy and availability	User demands to available all the time.
No hover state	No hover state may arise the problem of the wrong click.
Multi-device access	User's familiarity with multi-device access demands mental effort to understand layout.
Poor color contrast	It decreases the visibility of the screen.
Context-awareness	CAMA do not properly sense the environment and behave differently. This may cause user's disturbance and misunderstanding about the displayed information.
Wearable devices limited support	All the given wearable, smart glass devices and other accessories are facing the usability issues.



Usability Heuristics for Smartphone

- Visibility of the system status

- *Definition*

- The *smartphone* should keep the user informed about all processes and state changes through feedback and in a reasonable time. Feedback should facilitate user in understanding the device's behavior.

- *Explanation*

- All such changes should be visible and communicated to the user. There are some situations in which the user doesn't initiate the task, but system provides feedback in case of audio messages, low battery status and others. System provides feedback in different forms including sound, light color changes and others.

Visibility of the system status

- *Example:* Battery is showing the status that it is left 72%.
- *Benefits:*
 - Users feel comfort and react accordingly upon precise feedback. Users well aware about any application or system state changes within time limit.
- *Problems:*
 - There are many different types of feedback in which the user's familiarity with all these does matter. Message sound generates against text messages, notifications, Whatsapp messages/audio, and others. So, different sounds recognition and familiarity among other types of feedback become difficult to understand and differentiate.



Match between system and the real world

- *Definition*

- The smartphone should express users' language instead of system oriented designs and technicalities. The smartphone should present the information in a natural and balanced way. Smartphone should follow the real world pacts.

- *Explanation*

- Displayed Information is of images, text, icons, titles, font, style, and color scheme follow conventions and should present according to the situation. It is observed that due to non- effective color scheme, looks difficult for users to select any specific option/ icon. So, system should design keeping in view environment variance. Use of color scheme and combination should conform to community belief, values, culture and environment as well as to enhance UX.

Match between system and the real world

- *Example:*

Battery status and weather condition shows symbol of battery (bin) and clouds resembles respectively and according to the real world.

- *Benefits:*

Logical and natural arrangement of information supports every user to let it understand easily and perform the task efficiently.

- *Problems:*

Real world conventions vary due to different of beliefs, culture, conventions and others. So, possibly, due to subjective nature, this heuristic identifies a flaw but, that wouldn't be a flaw by some other community



Realistic error management

- *Definition*

- Users may not always be precise in performing the task on mobile devices due to many reasons, such as small screen size, kind of interaction etc. Application should provide mechanism to prevent from error as well as display error message with proper diagnosis and suggest suitable solution.

- *a: Explanation*

- System should provide proper security mechanism in case of sudden exit from the application domain, removing SIM/battery/sudden shutdown due to battery end up. In all such situations, system should provide a proper error message which highlights the consequences of the action and also provide a way to target situation in a good manner. When battery level goes down at a certain level, smartphone inform the user to take appropriate action such as enabling the power saving mode to save data and for long time usage.

- *b: Example*

- Smartphone displays error prevention mechanism, when the user wants to install or update application in case of low space. System also suggests appropriate solution to get free space in a good way as shown

Realistic error management

- *c: Benefits*

- Error prevention leads to enhance user comfort and less time to complete the task. Clear and recognizable error messages support the user to handle the situation just like a technician and recover from the error

- *d: Problems*

- This heuristic instructs the user to understand the nature and type of error and how to use the system to come out of the danger situation. All this demands to know the workflow of the system. This may be less helpful for the naïve users.



Help and Documentation

- The smartphone application should provide documentation and help focused on the user's current task.

- *a: Explanation*

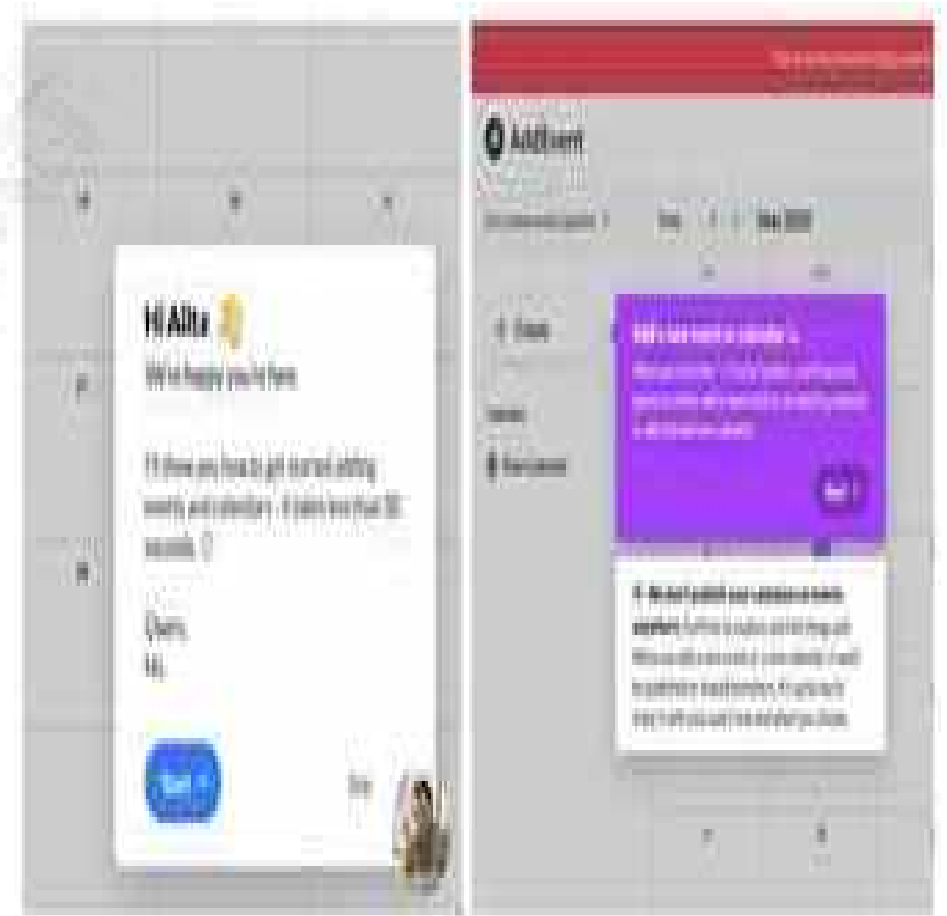
- Due to small screen size, designers put fewer information on screen. It may lead huge possibility to lose its meanings for the user. That's why, system should support the user to understand and execute the current task effectively and efficiently. One possible support is to provide proper help and documentation in an effective way. System should provide help and documentation with less cognitive work- load and context switching. Help and documentation may not necessarily organize in different menu setup or tab in mobile paradigm. System should support the user and provide help in the form of clues about next steps of task accomplishment.

- *b: Example*

- Smartphone provides proper help, when the user wants to display next image. Clear steps are mentioned to perform the current task. Another example is while using amazon mobile app, an option to add an item in the cart gives a tip to press and drag the item at specific place.

Help and Documentation

- *c: Benefits*
- Proper help supports the user to accomplish the task effectively and efficiently. This heuristic also provides support to the user to get complete understanding of the system and can execute task with less error.
- *d: Problems*
- This heuristic supports the user about how to execute the current task for proper accomplishment. So, for naïve user, it is relatively difficult to understand and recognize.



References

- <https://en.wikipedia.org/wiki/Usability>
- <https://nimitmangal.wordpress.com/2013/09/19/what-is-usability/>
- https://learn.org/articles/What_is_Usability_Engineering.html
- <https://ois2.ttu.ee> › uusois/USABILITYandUSER INTERFACE DESIGN

Software Construction and Development

Week-7

Efficiency of use and performance

- Different steps and concepts involve in defining the task accomplishment should contain the minimum set of requirements.
- **Explanation:**
- Smartphone should support the users to perform their respective tasks by providing refined and an abstracted interface to the user. System should explicitly provide basic application functionality, focused on users' objective. A task accomplishment requires system resources including processing capabilities, battery, number of steps, mental effort and others. Complex task requires comparatively more resources than a simple task.

Efficiency of use and performance

- Example
- If the user disconnects the call with “A”, after conversation. After some time, if the user dials to another person, let’s say “B”. Then, following steps are required to initiate the call. Press the dialed button, this results in opening up the previous caller interface (not necessary), from there, the user needs to go back with only “search” option. Another extra step is to cancel/delete the previously searched option, then find the contact number of “B” to call. Finally, press the dialed button. These extra steps further add up, if the dialer is busy in some other activity or moving.



Efficiency of use and performance

- Benefits

- Users comfort level goes up and intended to use the application in the future.

- Problems

- Possibly, one task accomplishment steps are minimum for an evaluator, while other raised question. So, it's a subjective in nature.

Aesthetic and minimalist design

- Dialogues should not contain information which is irrelevant or rarely needed.

- **Explanation**

- The application should provide aesthetically pleasant interaction, controls, icons, contents and others to the user. So, they can avoid negative UX and feel joyness while performing the task.

- **Example**

- An auto movement of cursor to the next field rather than scrolling enhances user experience. Similarly, when the user closes application, system should support the user in task accomplishment. But the displayed dialogue box contains additional and irrelevant information in the form advertisement, which enhances negative UX

Aesthetic and minimalist design

- Benefits
- Minimum and aesthetically pleasant interface leads to better performance and increase comfort level.
- Problems
- Different evaluators may have different interpretations due to the subjective nature of this heuristic.



Flexibility and efficiency of use

- System should provide shortcuts and configuration options for naïve and expert users to speed up interaction.
- **Explanation**
- Experienced users may prefer to configure gestures as per their own requirements (e.g., double tapping on a particular area for direct zoom-in a specific part of the map). Similarly, short keys are available for different operations to speed up interaction. Users can also customize various operations according to requirements. System also provide shortcut to mute all calls, stop vibration of incoming calls, alarms and timers, when the device is flipped.

Flexibility and efficiency of use

- **Example:**

Enabling the Wi-Fi by just flip the screen from the top and select the Wi-Fi icon. In the second way, Wi-Fi is turned on with some additional steps including “open the setting menu and then turning on Wi-Fi.

- **Benefits**

Users can accomplish the task efficiently and establish a sense of ownership.

- **Problems**

Each user has its own requirements and wants to customize accordingly. So, it is a difficult task for designers to accommodate all such users with full capacity.



Handling varied context of use in mobile environments

- Smartphone application should provide a familiar, natural and effective way to interact in different context of use. Interface interaction in all such constraints should be highly acceptable for users.
- **Explanation**
- Possible constraints are: auditory due to noise, visually due to bad light, shaking hands due to motion, or less focused due to social interaction. An effective interaction in these contexts of use facilitates the user to provide large controls, multi-modal input/output and subtle animation to accomplish task. System should also provide a usable setting of important functions of mobile application especially. It is observed that sometimes, due to movement, text is not readable on UI. It may be due to the use of improper color scheme and/or brightness restricts clear visibility, etc. System should support by putting a dot on some event/part of an application to improve the readability.

Handling varied context of use in mobile environments

- Example

By default, audio sound settings of ring tone is not audible in motion, due to noisy environment. Ring tone should be audible and further support with vibration can enhance UX.

- Benefits

Users perform the task efficiently in varied context of use.

- Problems

Evaluators may get confuse this heuristic with “match between system and the real world”.



Fingertip size controls and Ergonomics

- Smartphone should provide layout of the UI controls with enough margins, located at some recognizable places and fit to use.

- **Explanation**

While in motion, user mistakenly presses icon/button because of congestion which may lead to another unwanted situation. The system should provide enough margins to get fingertip size control. Holding smartphone with one hand and using the same hand thumb to interact with control elements will provide more ease to the user.



Fingertip size controls and Ergonomics

- Example

- “Save” and “Add” controls for events are very far from the single hand. So it requires second hand to save the event as shown in Figure, which is less ergonomically usable.

- Benefits

- User’s performance increases, when buttons are placed at some recognizable places and definitely easy to use.

- Problems

- One problem when executing this heuristic in the case of users with special needs. Evaluators should analyze each such case separately.

Effective Design to lessen the user's workload

- System should provide contextual information wherever necessary to the concerned application to provide usable interface to the user.
- **Explanation**
- As the user's objective is difficult to determine directly so context cues are used, to help infer this information and to inform an application for how best to support the user. While in motion, it is observed that users engage in multiple activities simultaneously and perform context switching according to the need. This requires more cognitive load to perform the task. Gestalt continuation law provides support to design interface, where one object link up to next object and so on. It supports the user to read properly and help to accomplish the task. For example, send message task has several steps like: open the app, write up the text, and enter/find the contact (receiver) and press the "send" button. In mobility, if these steps link up with each other, users can comfortably accomplish the task.

Effective Design to lessen the user's workload

- Example

- The map application in Figure supports users in traveling from one location to other by providing contextual information about the traffic condition, current place and routing information to lessen the workload.

- Benefits

- User gets support in task accomplishment and better understanding of the system.

- Problems

- Evaluators may get confuse in exercising this heuristic with match between system and the real world.

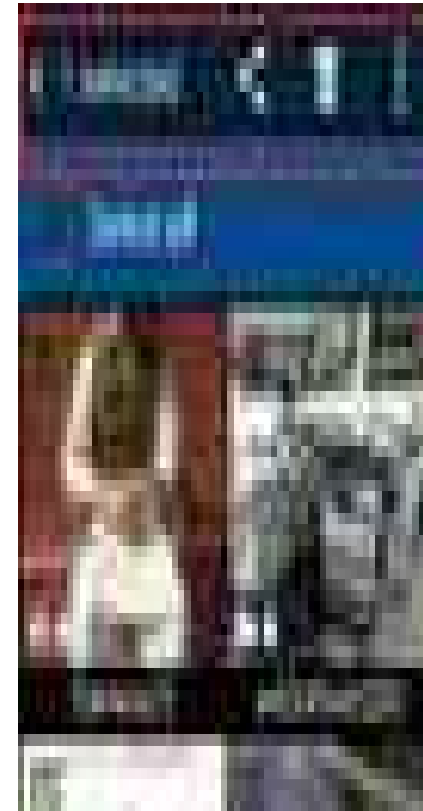


Recognition rather than recall

- The smartphone should offer visible actions, options and objects in order to avert operators to remember information from one part of the dialog to another. 
- **Explanation**
- System should provide recognizable multi-touch gestures while interacting with mobile application. Users while using double tap, slide, flick, two finger rotation, pinch in/out and many others, feel that these gestures will help to accomplish task in an effective manner. Make it sure to eliminate gesture ambiguity for significant gestures to be more usable according to touch targets, size and placement. Similarly, due to limited short term memory of users, system should also provide support in other parts.

Recognition rather than recall

- Example
- Delete icon (bin) is visible and recognizable for users
- Benefits
- Reduce mental effort and fewer chances for error.
- d: Problems
- Evaluators may get confuse in exercising this heuristic with” effective design to lessen the user’s workload”.



User control and obviousness

- System should facilitate users to protect information by providing undo/redo options and provide clearly an “emergency exits” to leave unwanted positions. These options should be available preferably through a physical button or equivalent.
- **Explanation**
- System should ensure user’s control and obviousness especially in modification/deletion of any important information. Users can easily manage the application and resource consumption conveniently. System should support users to navigate comfortably across the application.
- **Example**
- Placing a pattern or fingerprint identification makes it possible that no unauthorized person can access any personal information

User control and obviousness

- Benefits
 - Users feeling of ownership can enhance UX and they can perform the task efficiently.
- Problems
 - This heuristic provides an insight to repair an error and provides control to users to let it work with their own needs. So, this heuristic may not overlap with “flexibility and efficiency of use”.



Consistency and standards

- The smartphone should follow the well-known conventions, allowing the user to do things in a familiar, standard and consistent way.
- **Explanation**
- Every application contains similar functions. These similar functions put together with same layout design. This is the reason; users feel comfort and efficiently perform the task. Same options can deploy at different UIs of mobile application. These should also have same designed mechanics to provide easy to use interaction. These conventions and standards should also shadow in different devices of same manufacturers.

Consistency and standards

- Example

- For example, Gallery mobile application contains pictures, where we can edit/modify/insert/delete any image.

- Benefits

- Consistent options help users to get familiarity, easy to learn and memorization. This will certainly increase task efficiency, minimize errors and increase user satisfaction.

- Problems

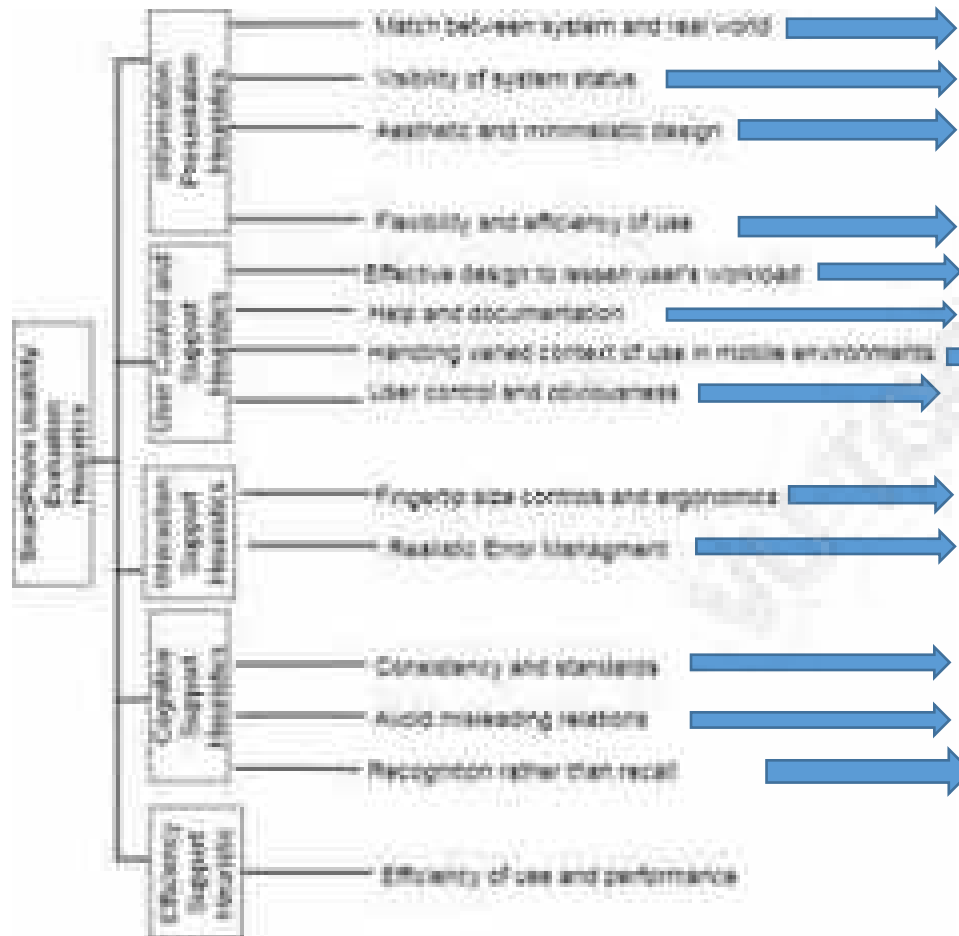
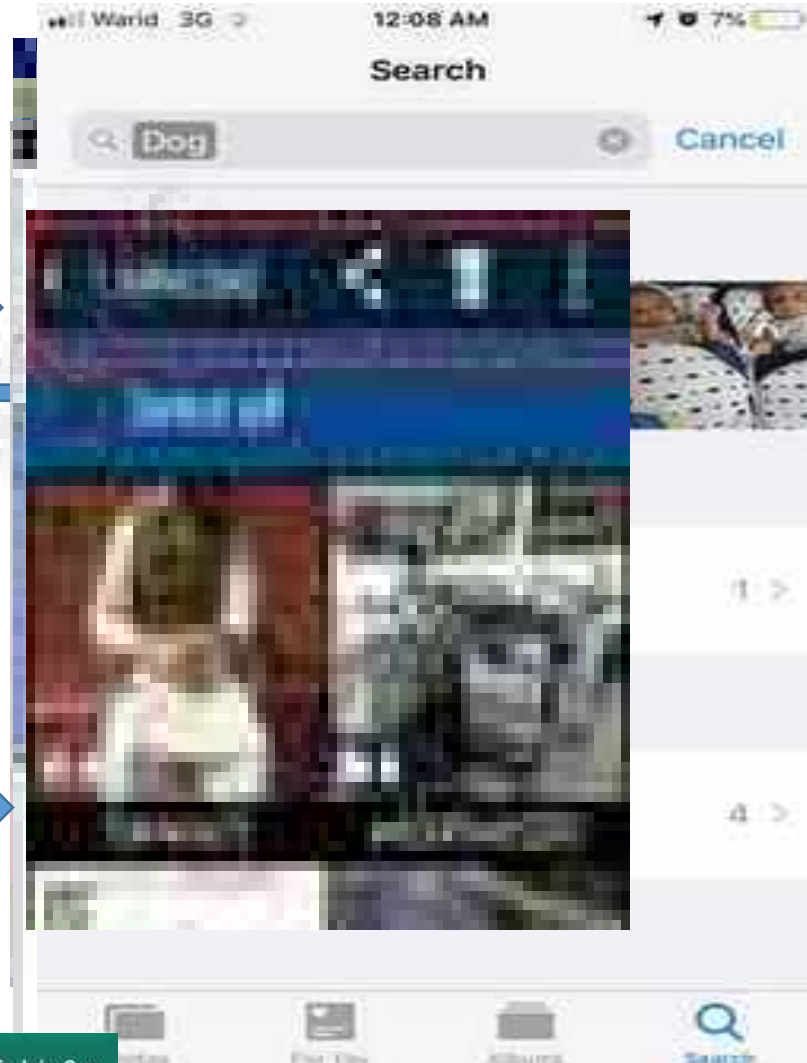
- Evaluators may get confuse in exercising this heuristic with “match between system and the real world.



Framework for Classification of Usability heuristics

- **Information Presentation features:** This category contains features associated with the smartphone presentation and information.
- **Interaction features:** This category deals with the interaction between the user and smartphone.
- **User's Cognitive features:** Category associated with the user's mental model/perception to perform any specific task.
- **User's Control and Support:** This category contains features that provide control and support to the user during and after completion of any specific task.
- **Efficiency related features:** This category contains aspects associated with the efficiency of the task accomplishment.

Framework for Classification of Usability heuristics



Software Construction and Development

Week-8

Object Oriented Design Principles

- **Information Hiding**

- Information is stored within the object
- It is hidden from the outside world
- It can only be manipulated by the object itself
- Example – Name of a Person, can't read the numbers directly from the SIM

- **Encapsulation**

- Data and behavior are tightly coupled inside an object
- Information structure and implementation of its operations are hidden from the outer world

Example: A Phone stores numbers in digital format and knows how to convert it into human-readable characters

Object Oriented Design Principles

- Object has an Interface

- An object encapsulates data and behaviour, So how objects interact with each other?
- Each object provides an interface (operations), Other objects communicate through this interface
- Example: Interface of a Car (Steer Wheels, Accelerate, Change Gear, Apply Brakes, Turn Lights On/Off)

- Abstraction

- Abstraction is a way to cope with complexity.
- “Capture only those details about an object that are relevant to current perspective”
- Example: Ahmad is a Student (StdId, StName, GPA etc..) Ali is a PhD student
- Ali is a PhD student and tea



Object Oriented Design Principles

- Inheritance:

- A child inherits characteristics of its parents, a child may have its own unique characteristics.
- The parent class is called *base class* and the child class is called *derived class*,
- Inheritance – “IS A” or “IS A KIND OF” Relationship
- Example – Person is a parent class and student, teacher are derived classes
- Reuse
- Concepts Related with Inheritance

- Generalization (classes may have common characteristics, extract

features into a new class and use it as a base class. Example line, circle, rectangle into shape class

Object Oriented Design Principles

- Association

- An object keeps associations with other objects to delegate tasks

- Class Association

- Inheritance

- Object Association

- Simple Association (weakest link, e.g. Ali lives in a House)
- Composition (An object may be composed of other smaller objects, “part” objects and the “whole”, e.g. fan has wings)

Object Oriented Design Principles

- Polymorphism

- Overriding (Area function in shape overrides in Circle, Triangle and Rectangle)
- Abstract Class (Shape and Person)
- Concrete Classes
- Polymorphism refers to existence of different forms of a single entity
- Example, Diamond and Coal are different forms of Carbon
- polymorphism means that different objects can behave in different ways for the same message (stimulus)
- Consequently, sender of a message does not need to know exact class of the receiver
- Example: Area Function

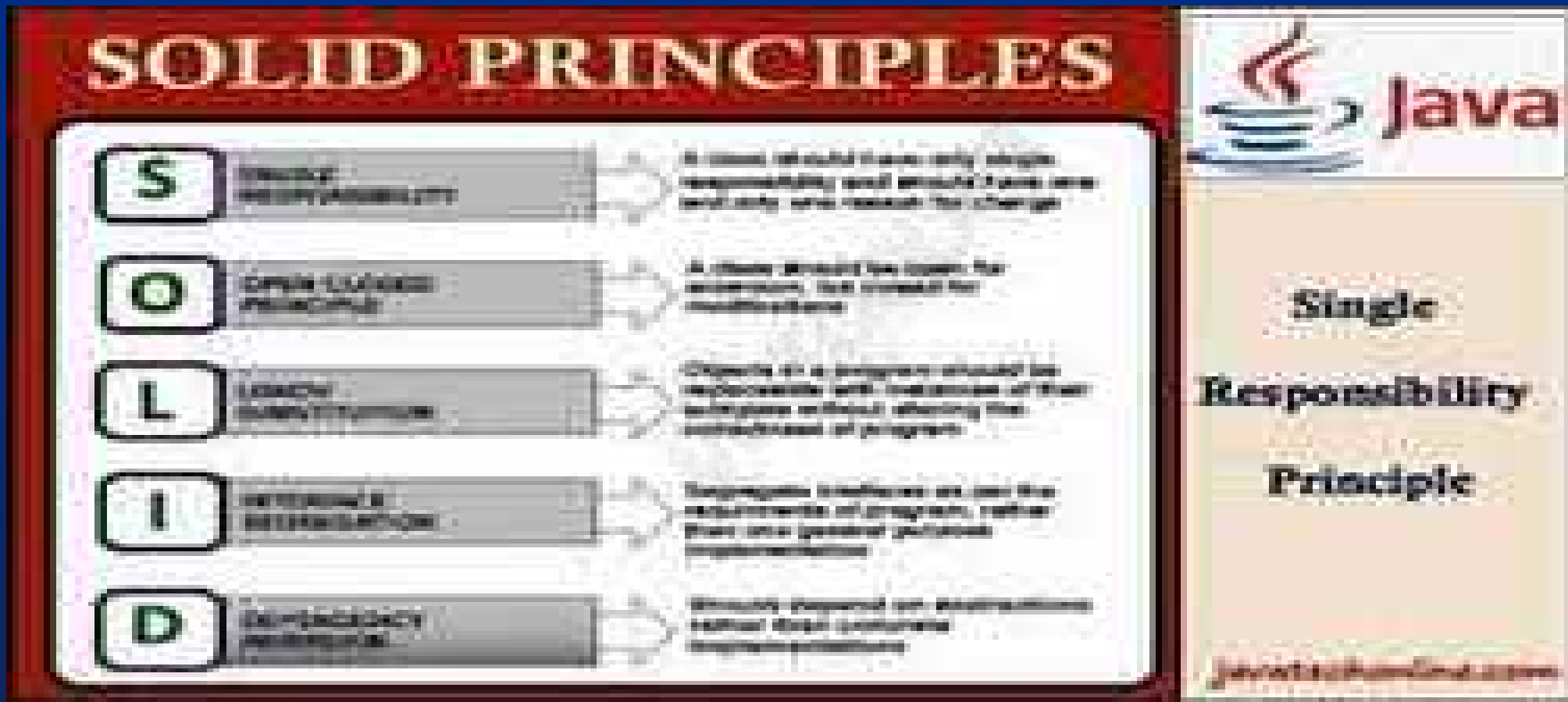


Why need of Software Design Principles?

- To avoid 3 important characteristics of a bad design

- Rigidity - It is hard to change because every change affects too many other parts of the system.
- Fragility - When you make a change, unexpected parts of the system break.
- Immobility - It is hard to reuse in another application because it cannot be disentangled from the current application.

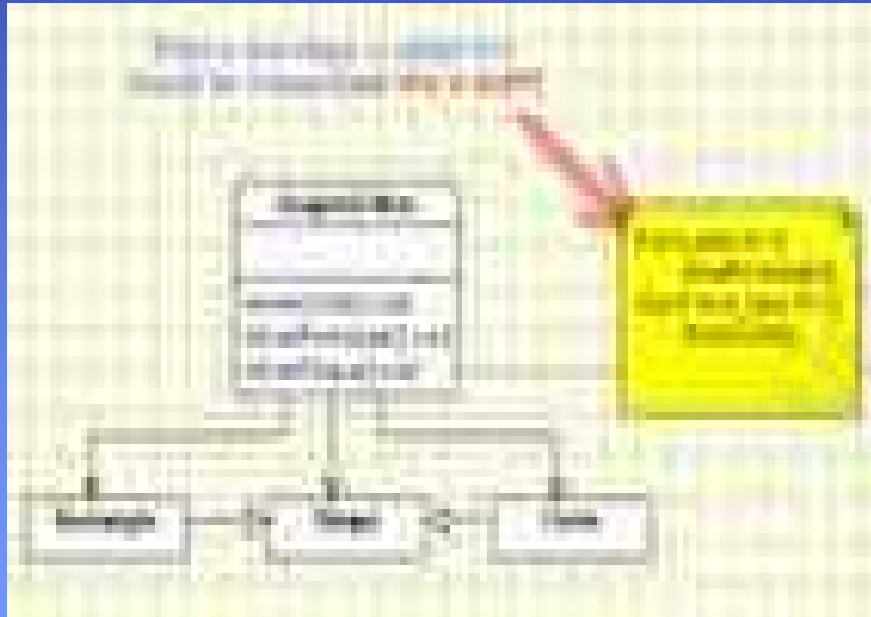
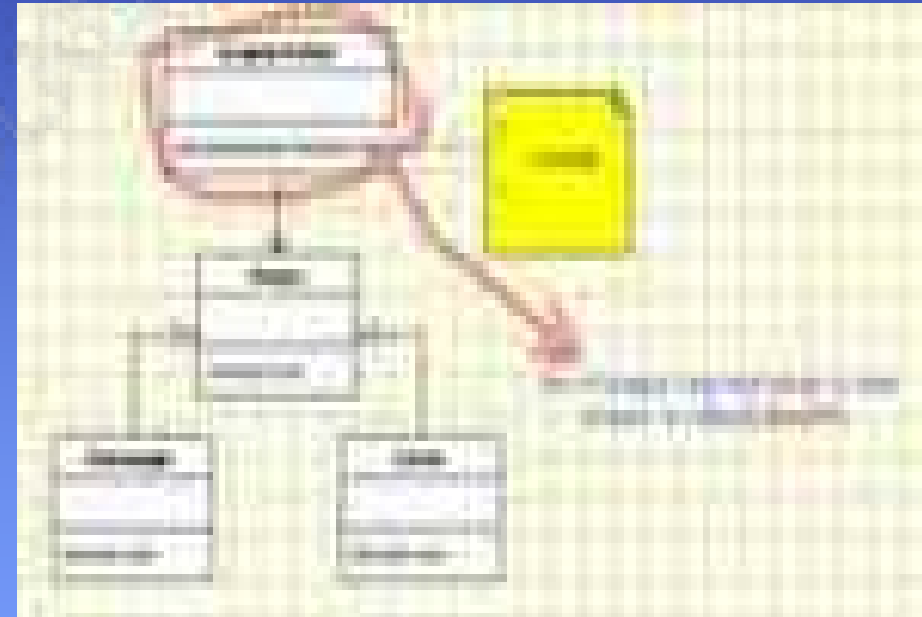
SOLID: Object Oriented Design Principles



Open Close Principle

- Software entities like classes, modules and functions should be open for extension but closed for modifications.
- OC is a generic principle.
- The same principle can be applied for modules, packages, libraries.
- When referring to the classes Open Close Principle can be ensured by use of Abstract Classes and concrete classes for implementing their behavior.
- This will enforce having Concrete Classes extending Abstract Classes instead of changing them.
- Some particular cases of this are Template Pattern and Strategy Pattern.

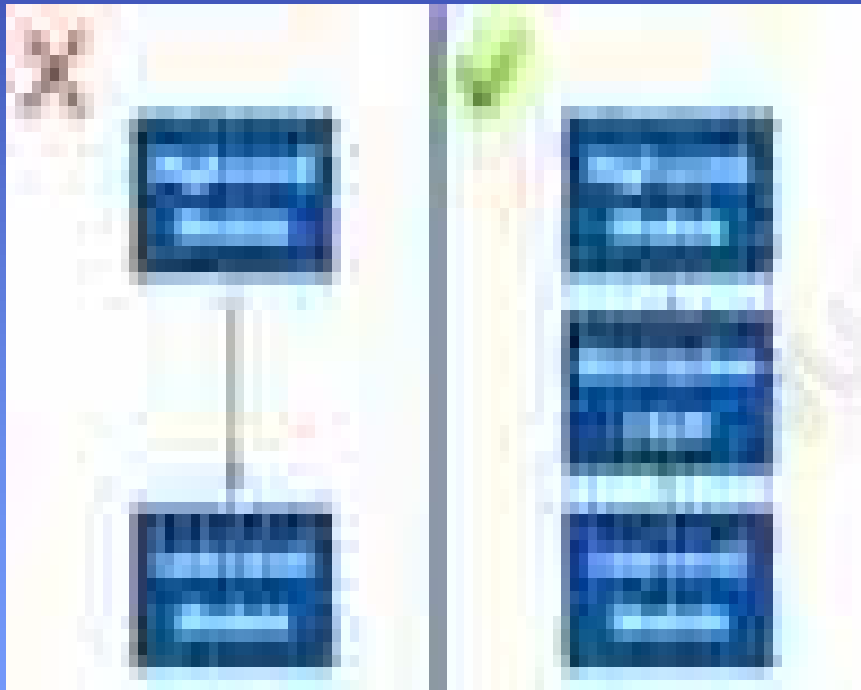
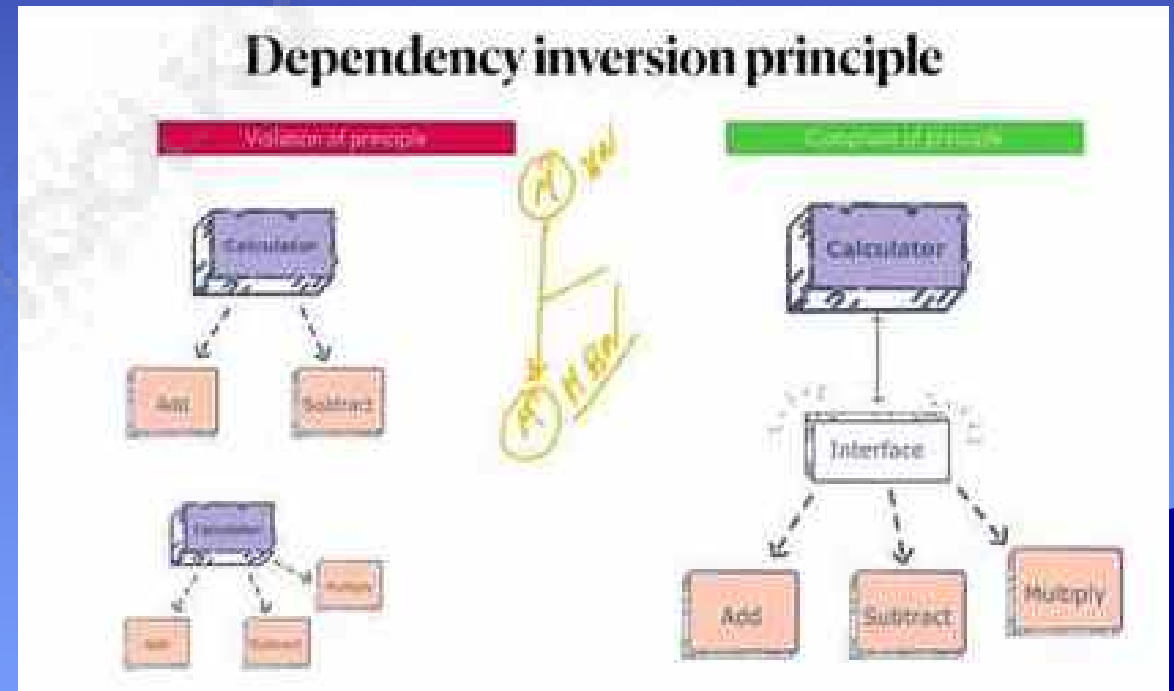
Open Close Principle



Dependency Inversion Principle

- High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.
- Dependency Inversion Principle states that we should decouple high level modules from low level modules, introducing an abstraction layer between the high level classes and low level classes.
- Further more it inverts the dependency: instead of writing our abstractions based on details, the we should write the details based on abstractions.
- By applying the Dependency Inversion the modules can be easily changed by other modules just changing the dependency module.

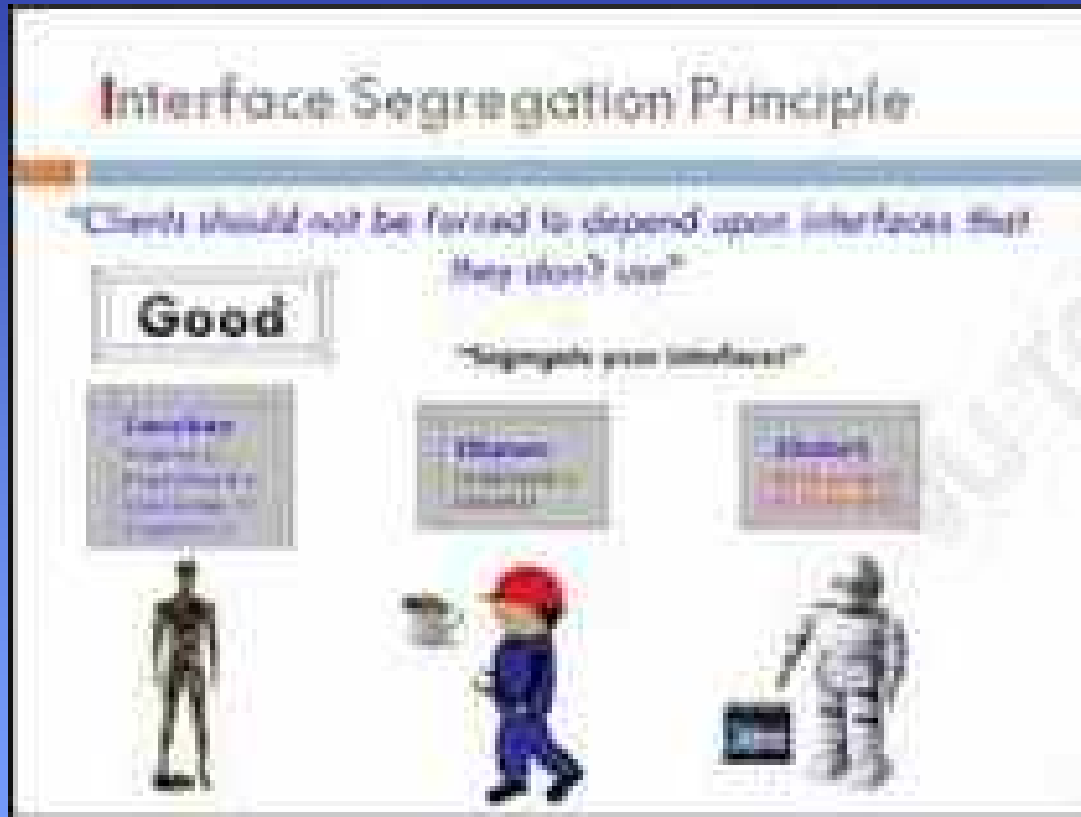
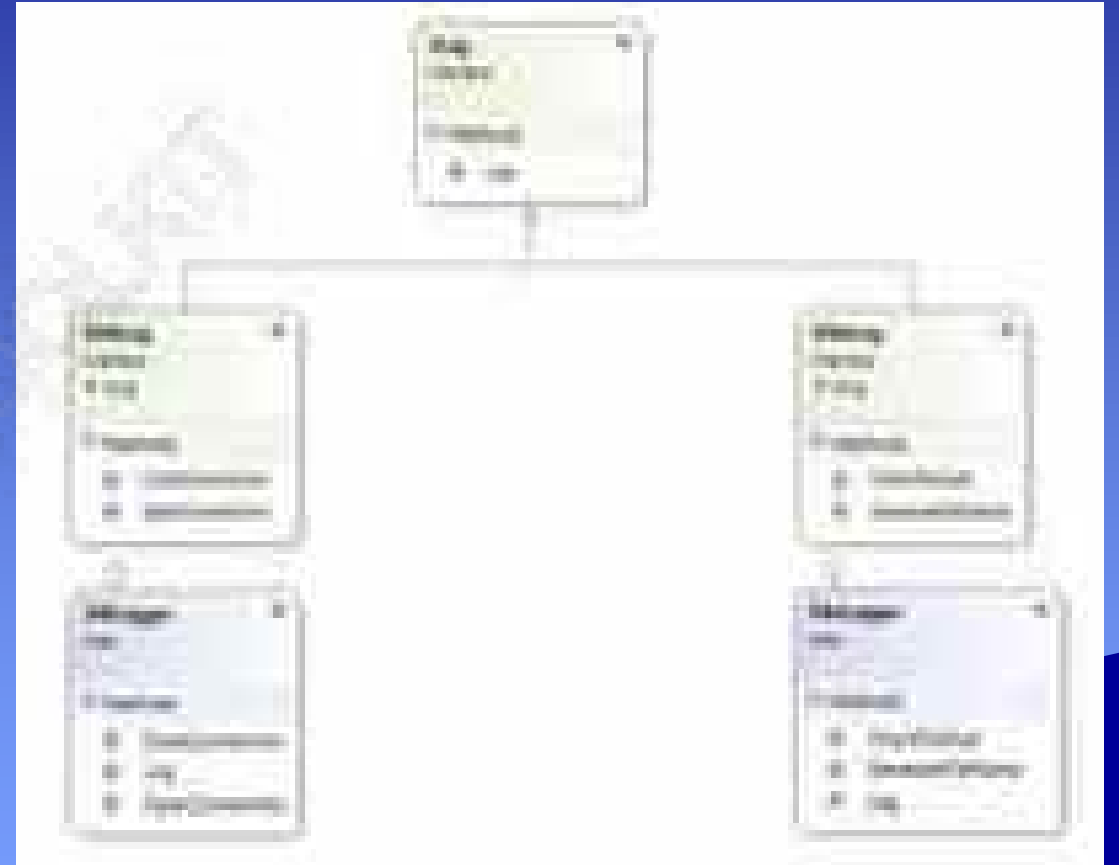
Dependency Inversion Principle



Interface Segregation Principle

- Clients should not be forced to depend upon interfaces that they don't use.
- This principle teaches us to take care how we write our interfaces. When we write our interfaces we should take care to add only methods that should be there. If we add methods that should not be there the classes implementing the interface will have to implement those methods as well. For example if we create an interface called Worker and add a method lunch break, all the workers will have to implement it. What if the worker is a robot?

Interface Segregation Principle



Single Responsibility Principle

- A class should have only one reason to change.
- In this context a responsibility is considered to be one reason to change. This principle states that if we have 2 reasons to change for a class, we have to split the functionality in two classes. Each class will handle only one responsibility and on future if we need to make one change we are going to make it in the class which handle it. When we need to make a change in a class having more responsibilities the change might affect the other functionality of the classes.

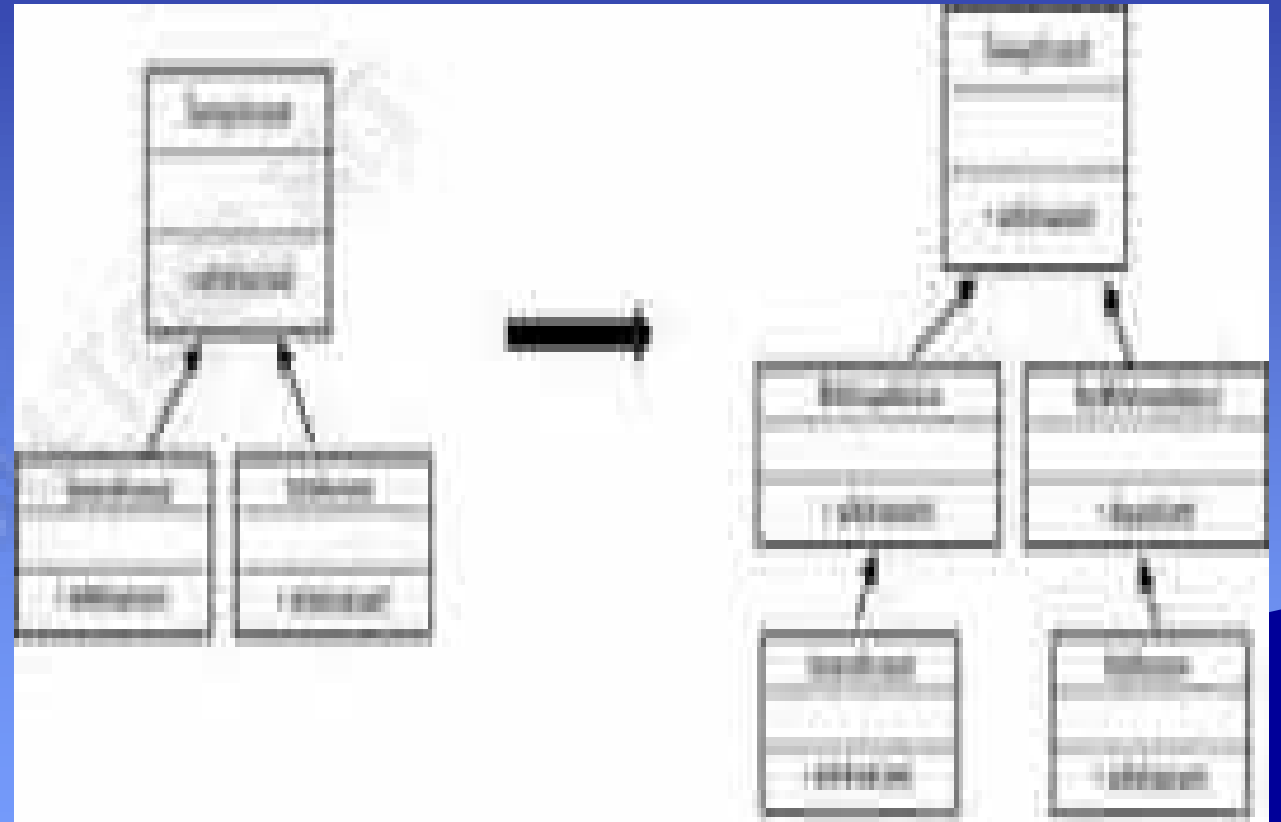
Single Responsibility Principle



Liskov's Substitution Principle

- Derived types must be completely substitutable for their base types.
- This principle is just an extension of the Open Close Principle in terms of behavior meaning that we must make sure that new derived classes are extending the base classes without changing their behavior. The new derived classes should be able to replace the base classes without any change in the code.

Liskov's Substitution Principle



Principle of least knowledge

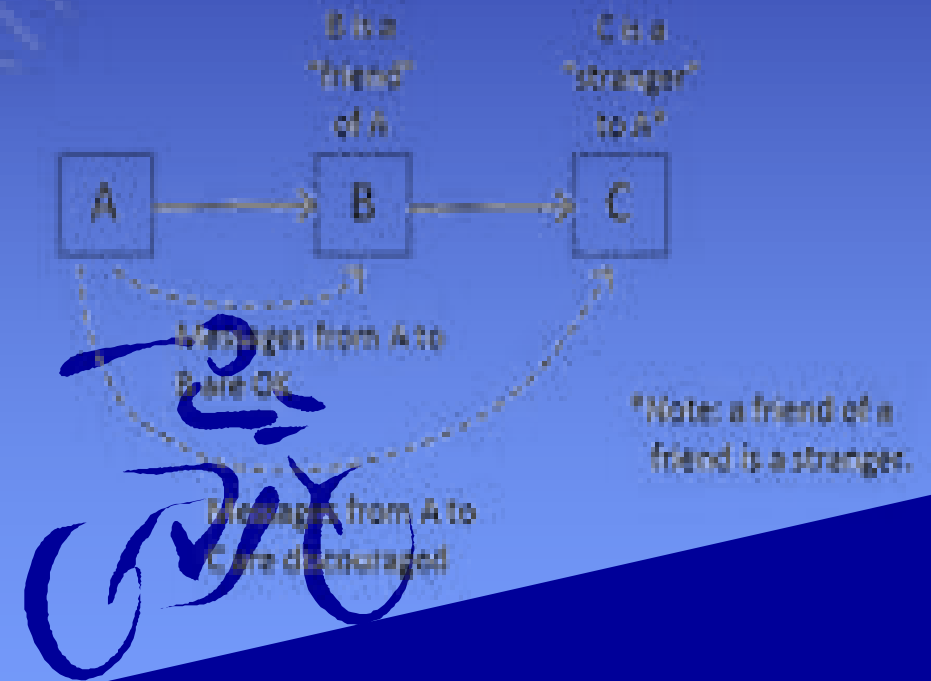
- The Law of Demeter (LoD) or principle of least knowledge is a design guideline for developing software, particularly object-oriented programs.
- Demeter's law is known as “don't talk to strangers” because:
 - Each unit should have only limited knowledge about other units — only units “closely” related to the current unit.
 - Each unit should only talk to its friends — don't talk to strangers.

Principle of least knowledge

- More formally, the Law of Demeter requires that a method *m* of an object *O* may only invoke the methods of the following kinds of objects:

- *O* itself.
- *m*'s parameters.
- Any objects instantiated within *m*.

- *O*'s direct component objects.



Object-Oriented Modeling

An Example



Problem Statement

- **Develop a graphic editor that can draw different geometric shapes such as line, circle and triangle. User can select, move or rotate a shape. To do so, editor provides user with a menu listing different commands. Individual shapes can be grouped together and can behave as a single shape.**



Identify Classes

➤ Extract nouns in the problem statement

- Develop a graphic editor that can draw different geometric shapes such as line, circle and triangle. User can select, move or rotate a shape. To do so, editor provides user with a menu listing different commands. Individual shapes can be grouped together and can behave as a single shape.



...Identify Classes

- Eliminate irrelevant classes
- ▶ Editor – Very broad scope
- ▶ User – Out of system boundary



...Identify Classes

➤ Add classes by analyzing requirements

- **Group – required to behave as a shape**
 - “Individual shapes can be grouped together and can behave as a single shape”
- **View – editor must have a display area**

...Identify Classes

➤ Following classes have been identified:

- Shape

- Line

- Circle

- Triangle

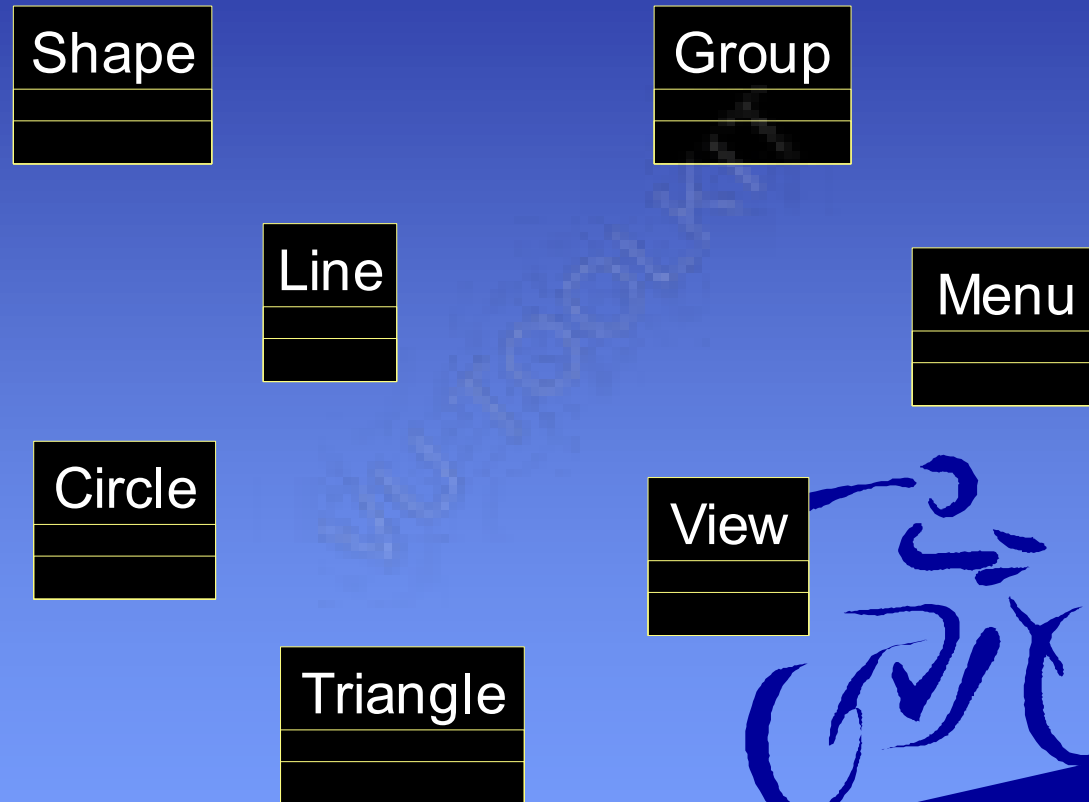
- Menu

- Group

- View



Object Model – Graphic Editor



Identify Associations

➤ Extract verbs connecting objects

- “Individual shapes can be grouped together”
 - Group consists of lines, circles, triangles
 - Group can also consists of other groups
(Composition)



... Identify Associations

➤ Verify access paths

- **View contains shapes**
 - View contains lines
 - View contains circles
 - View contains triangles
 - View contains groups
(Aggregation)



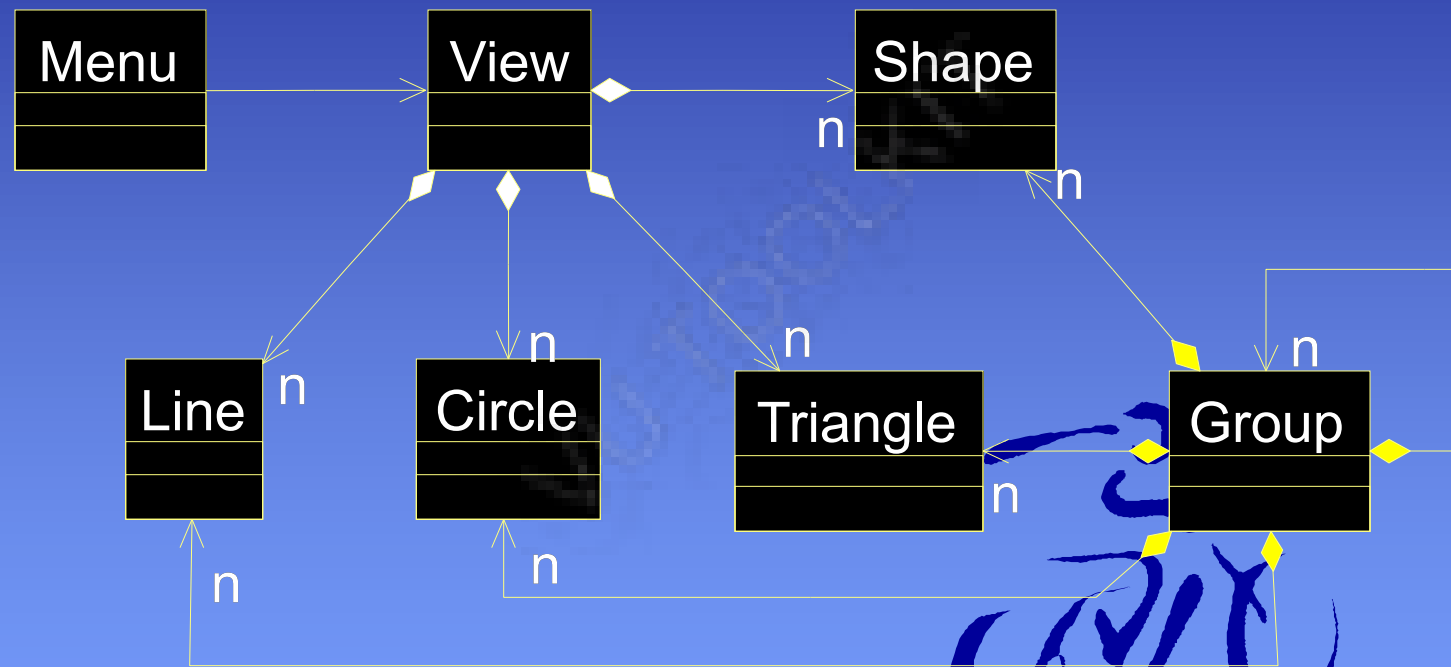
... Identify Associations

➤ Verify access paths

- **Menu sends message to View**
(Simple One-Way Association)



Object Model – Graphic Editor



Identify Attributes

- Extract properties of the object
 - From the problem statement
- ▶ Properties are not mentioned



...Identify Attributes

- Extract properties of the object
 - From the domain knowledge

- **Line**

- Color
- Vertices
- Length

- **Circle**

- Color
- Vertices
- Radius

- **Triangle**

- Color
- Vertices
- Angle

- **Shape**

- Color
- Vertices



...Identify Attributes

- Extract properties of the object
 - From the domain knowledge

- **Group**

- noOfObjects

- **View**

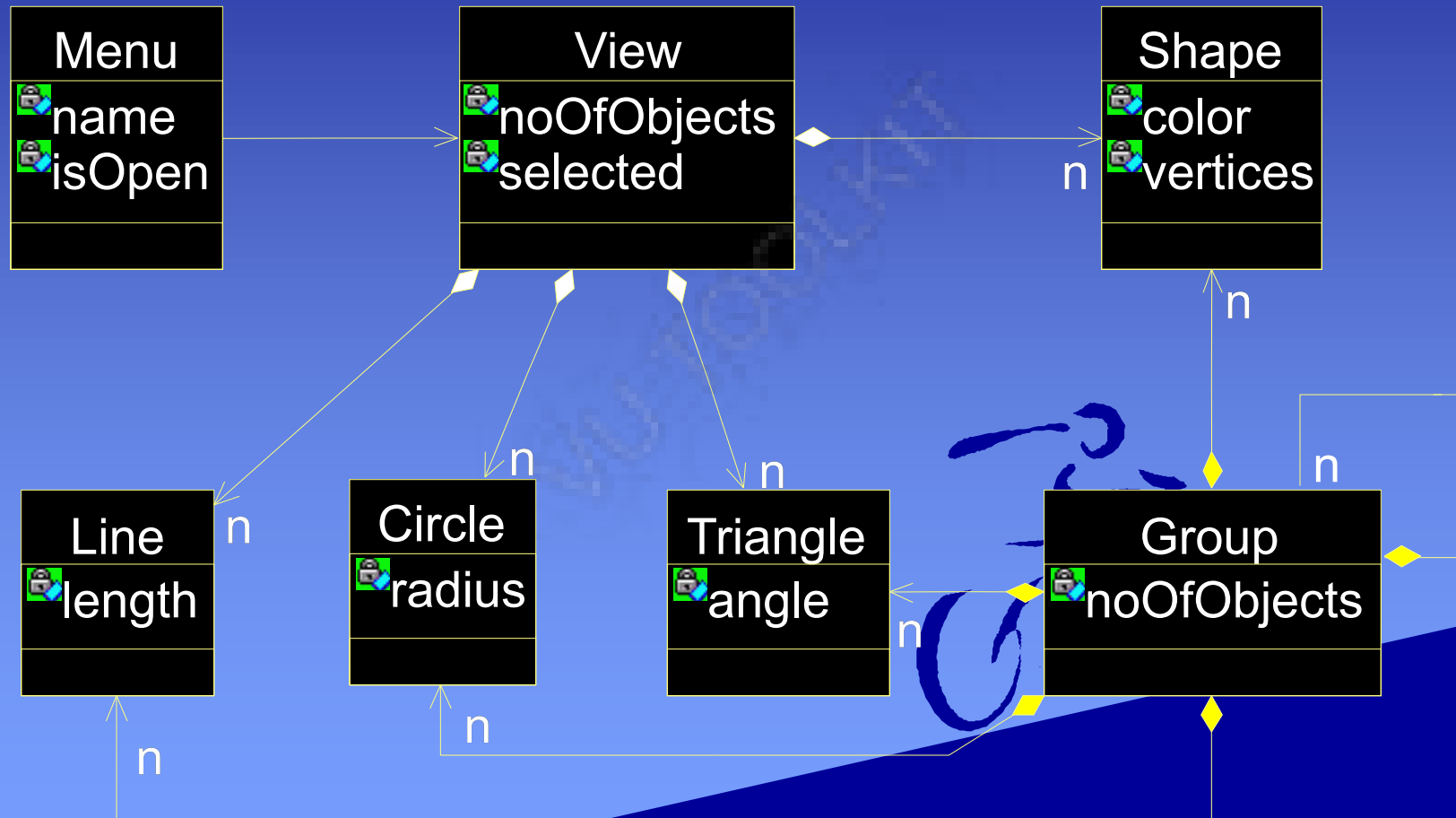
- noOfObjects
 - selected

- **Menu**

- Name
 - isOpen



Object Model – Graphic Editor



Identify Operations

- Extract verbs connected with an object
- **Develop** a graphic editor that can **draw** different geometric shapes such as line, circle and triangle. User can **select**, **move** or **rotate** a shape. To do so, editor **provides** user with a menu listing different commands. Individual shapes can be **grouped** together and can **behave** as a single shape.

... Identify Operations

- Eliminate irrelevant operations
- ▶ Develop – out of system boundary
- ▶ Behave – have broad semantics



...Identify Operations

➤ Following are selected operations:

- **Line**

- Draw
- Select
- Move
- Rotate

- **Circle**

- Draw
- Select
- Move
- Rotate



...Identify Operations

➤ Following are selected operations:

- **Triangle**

- Draw
- Select
- Move
- Rotate

- **Shape**

- Draw
- Select
- Move
- Rotate



...Identify Operations

➤ Following are selected operations:

- **Group**

- Draw
- Select
- Move
- Rotate

- **Menu**

- Open
- Select
- Move
- Rotate



...Identify Operations

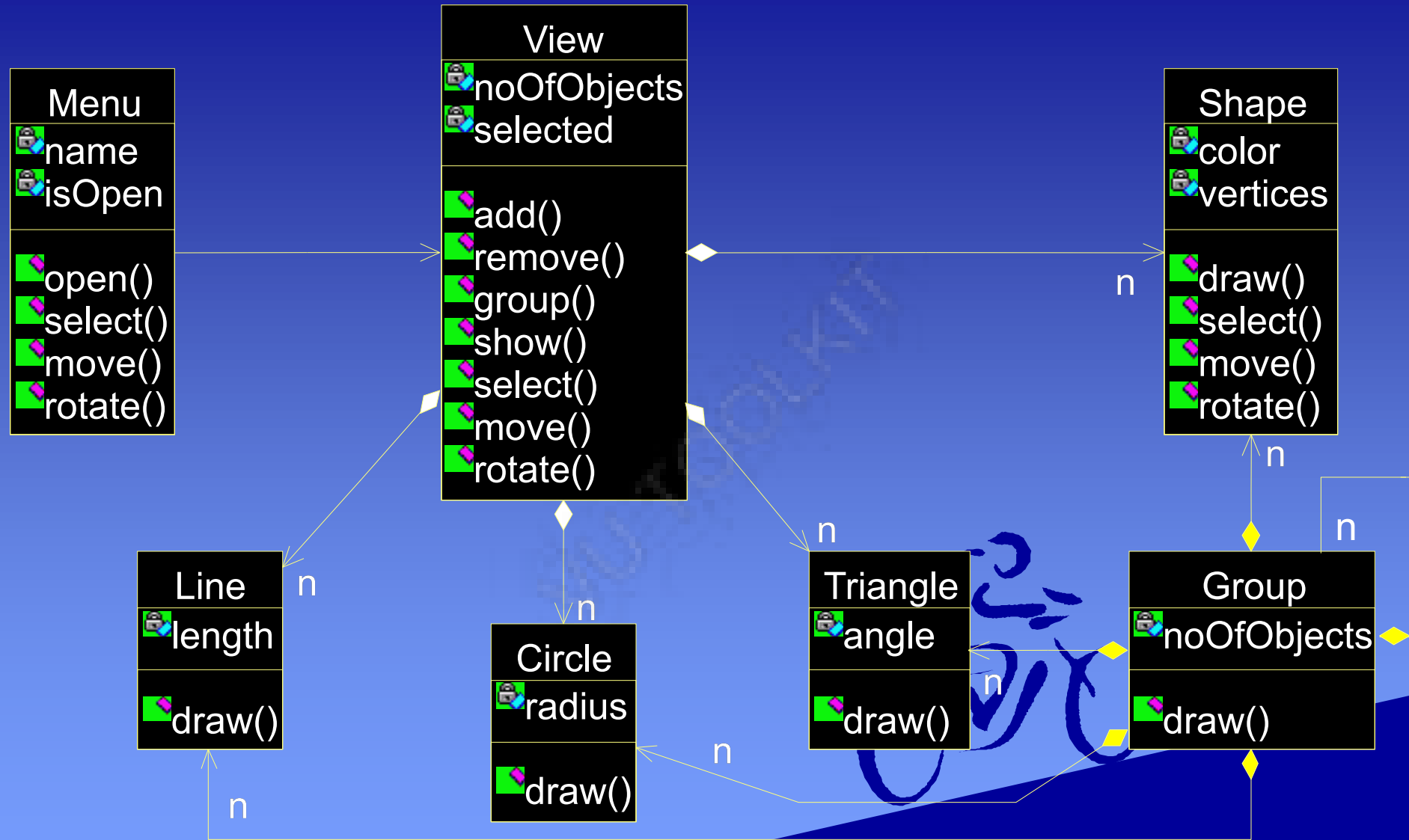
➤ Extract operations using domain knowledge

- **View**

- Add
- Remove
- Group
- Show

- Select
- Move
- Rotate





Identify Inheritance

➤ Search “is a kind of” by looking at keywords like “such as”, “for example”, etc

- “...shapes such as line, circle and triangle...”
 - Line, Circle and Triangle inherits from Shape



...Identify Inheritance

➤ By analyzing requirements

- “Individual shapes can be grouped together and can behave as a single shape”
 - Group inherits from Shape



Refining the Object Model

- Application of inheritance demands an iteration over the whole object model
- In the inheritance hierarchy,
 - All attributes are shared
 - All associations are shared
 - Some operations are shared
 - Others are overridden



...Refining the Object Model

➤ Share associations

▶ View contains all kind of shapes

▶ Group consists of all kind of shapes



...Refining the Object Model

➤ Share attributes

- Shape – Line, Circle, Triangle and Group
 - Color, vertices



...Refining the Object Model

➤ Share operations

- Shape – Line, Circle, Triangle and Group

- Select
- Move
- Rotate



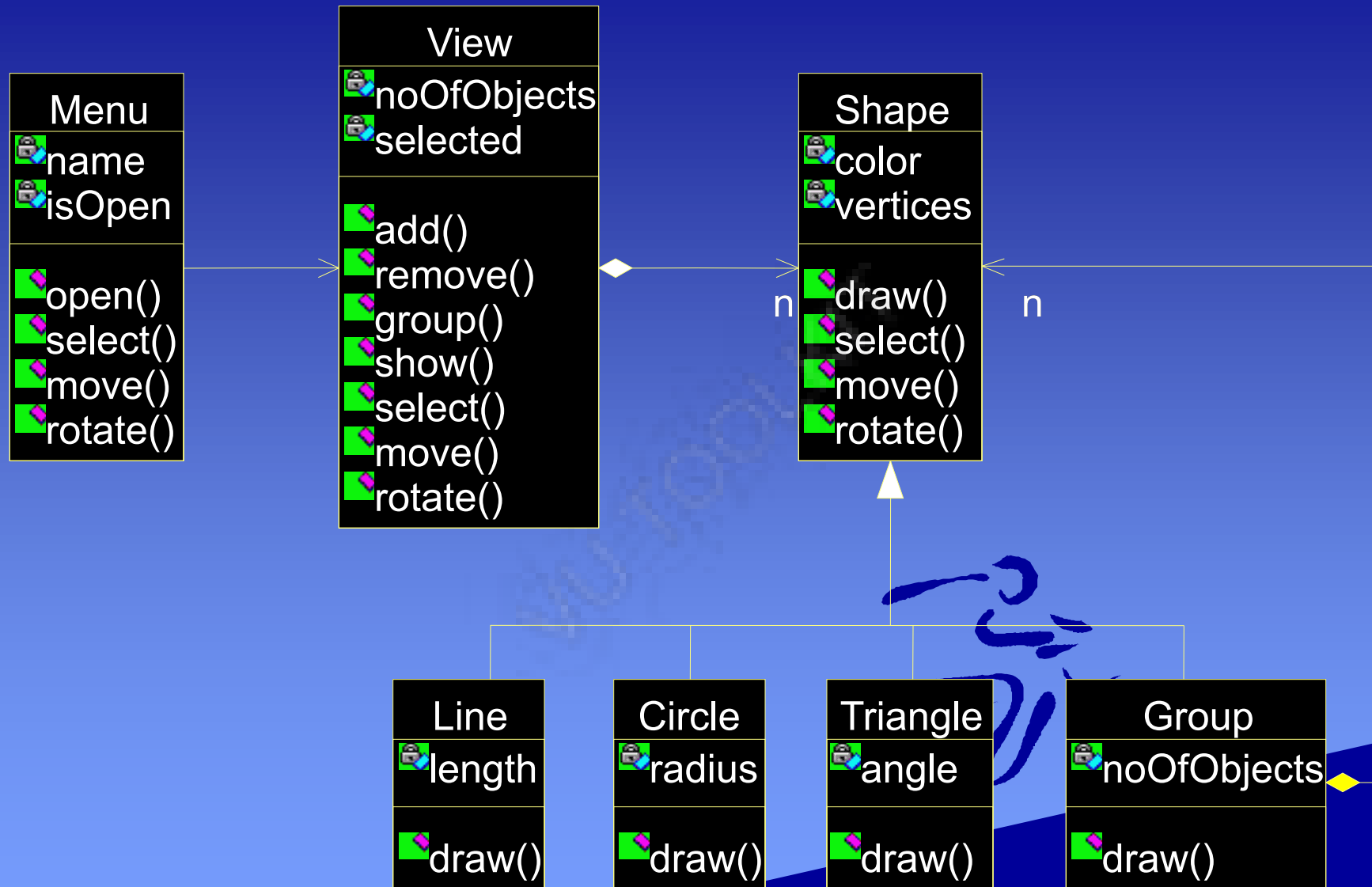
...Refining the Object Model

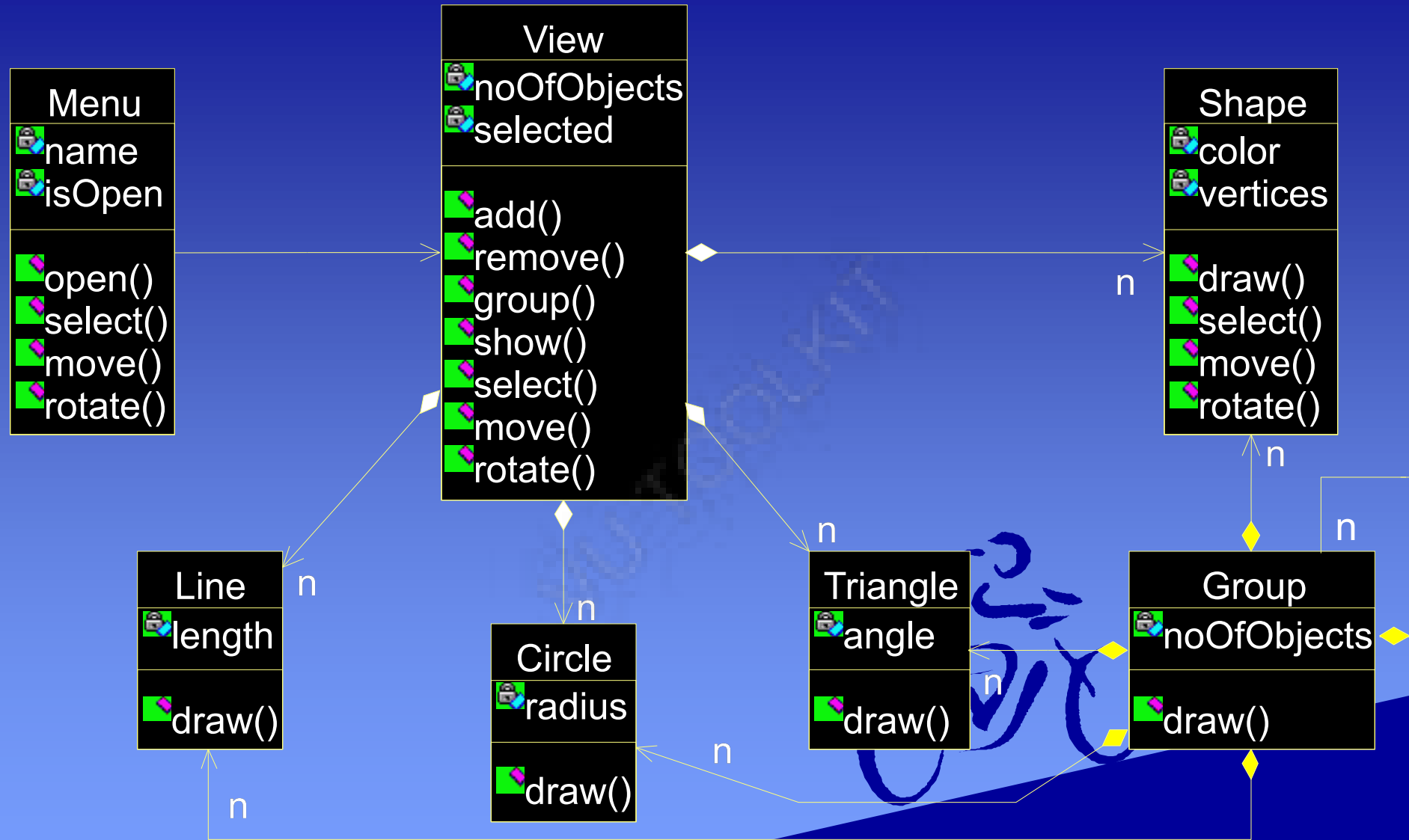
➤ Share the interface and override implementation

- Shape – Line, Circle, Triangle and Group

- Draw







Week 9

Coding Standard/Guidelines



What is a coding standard

- A set of guidelines that recommend programming style, practices, and methods for each aspect of a program
- Usually covers
 - Naming Conventions : Variable, Function , Class Names, File Names
 - White spaces
 - Indentation, Bracket Placement
 - Comments
 - etc. etc..

Why need Coding Standards?

- Help improve the readability of the source code
- Make software maintenance easier
- Less Bugs
- Team Work
- Reduces the cost of maintenance
- Task Automation (helps the scripts to read code better)

File Naming

- For every project, a file named Driver.cpp containing only the main() function is required.
- Executable should be named driver.out
- All C++ classes are defined in a separate header file.
 - File name should be Classname.h
 - Implement only getters/setters in the header
- All C++ classes are implemented in a separate source code file
 - File name should be Classname.cpp
 - Implement all the other required functions here

Class Definition Standards

- All class names begin with uppercase
- Only one private, protected and public section in a class definition.
 - public comes first, followed by protected and then private.
- Class methods must have complete function header comments
 - Except getters/setters
 - Class methods must be const whenever possible
- Class data members name **MAY** begin with m_
 - `int m_length;`

Variable & Function Naming

- Use meaningful descriptive variable names
 - float radius instead of float r;
- Use lowerCamel case for variables
 - int numStudents instead of int num_of_students
- Function MAY begin names with uppercase letters
 - GetName() or getName()
- function prototypes must include parameter names as well as types
- Default values for parameters must be specified in the prototype

Avoid Magic Numbers

- Constants should be used for magic numbers and strings whenever possible
 - `const double PI = 3.14159`

Wrong

```
int array[30];  
  
for (int i = 0; i < 30; i++)  
    array[i] = -3;  
  
if (command == "ADD")  
    // do something  
else if (command == "MULTIPLY")  
    // do something else  
  
// A magic number  
// here it is again  
// what does -3 mean?  
  
// magic string may change someday  
// another magic string
```

Right

```
const int ARRAYSIZE = 30;  
const int INITIALVALUE = -3;  
const string ADD = "ADD";  
const string MULTIPLY = "MULTIPLY";  
  
int array(ARRAYSIZE);  
  
for (int i = 0; i < ARRAYSIZE; i++)  
    array[i] = INITIALVALUE;  
  
if (command == ADD)  
    // do something  
else if (command == MULTIPLY)  
    // do something else
```

Indentation, Spacing

- Use blank lines to separate pieces of code for readability to separate unrelated sections of code

Wrong

```
while (moreStudents == true) {  
    // read student last name  
    ReadStudentName();  
    // print full name  
    PrintStudentName();  
    // now verify student info  
    goodInfo = VerifyStudentInfo(student);  
    if (goodInfo == true)  
        cout << "good info" << endl;  
}
```

Right

```
while (moreStudents == true)  
{  
    // read student last name  
    ReadStudentName();  
  
    // print full name  
    PrintStudentName();  
  
    // now verify student info  
    goodInfo = VerifyStudentInfo(student);  
    if (goodInfo == true)  
        cout << "good info" << endl;  
}
```

Indentation, Spacing

- Use a **tab** or **4** spaces for each level of indentation
- Use spaces around all operators.

Wrong

```
x=y+8;
```

```
If (x>y)
{statement1;}
else
    {statement 2;}
```

Right

```
x = y + 8;
```

```
If (x > y)
{
    statement1;
}
else
{
    statement 2;
}
```

Braces Recommended Styles

K & R Style

```
int power(int x, int y)
{
    int result;

    if (y < 0) {
        result = 0;
    } else {
        result = 1;
        while (y-- > 0) {
            result *= x;
        }
    }

    return result;
}
```

Allman/BSD Style

```
int power(int x, int y)
{
    int result;

    if (y < 0)
    {
        result = 0;
    }
    else
    {
        result = 1;
        while (y-- > 0)
        {
            result *= x;
        }
    }

    return result;
}
```


Brace Usage

- Use a consistent style for braces
- Braces are required for single statement if/else/while/for structures

Wrong

```
if (grade > 99)
    cout << "You got an A" << endl;
else
    cout << "No A for you" << endl;

for (i = 0; i < 30; i++)
    array[i] = -3;
```

Right

```
if (grade > 99)
{
    cout << "You got an A" << endl;
}
else
{
    cout << "No A for you" << endl;
}

for (i = 0; i < 30; i++)
{
    array[i] = -3;
}
```

Comments

- Comments are the programmer's main source of documentation.
- **Rule of Thumb:**
 - Approximately every 5 lines of code need AT LEAST 1 comment.
 - Not every line of code needs a comment.
 - Constant declarations MUST have 1 comment.
- Every cpp and .h file should contain File header comment
- Every function should contain function header comment.

File Header Comments

EVERY .cpp and .h file should contain an opening comment describing the contents of the file And **MUST** include the following information.

- The file name
- The project number
- Your name
- The date the file was created
- Your section number
- Your tamu e-mail address
- A description of what the code in the file does

```
/* ****  
** File      : Driver.cpp  
** Project   : CSCE 221 Project 1, Spring 2019  
** Author    : Bob Smith  
** Date      : 1/22/19  
** Section   : 0104  
** E-mail    : bsmith32@tamu.edu  
**  
** This file contains the main driver program for Project 1.  
** This program reads the file specified as the first command line  
** argument, counts the number of words, spaces, characters, and  
** displays the results in the format specified in the project description.  
**  
**** */
```



Function Header Comments

- Each FUNCTION and CLASS METHOD (Except getters/setters) must have a header comment that includes the following information
 1. The function's name
 2. The function's **pre-condition(s)** (if there are no pre-conditions, say so).
 3. The function's **post-condition(s)**.

Circle.h

```
//-----  
// Name: CircleArea  
// PreCondition: the radius is greater than zero  
// PostCondition: Returns the calculated area of the circle  
//-----  
double CircleArea (double radius );
```

Circle.cpp

```
// CircleArea  
// Given the radius, returns the area  
double CircleArea ( double radius )  
{  
    const double PI = 3.14159;  
  
    // handle unmet precondition  
    if (radius < 0.0)  
        return 0.0;  
    else  
        return PI * radius * radius;  
}
```

Inline Comments

- MUST appear above the code to which it applies and is indented to the same level as the code

Wrong

```
for (i = 0; i < ARRAYSIZE; i++) // check every element of the array
{
    if (array[i] % 2 == 1) // add one to odd values
    {
        array[i]++;
    }
}
```

Right

```
// check every element of the array
for (i = 0; i < ARRAYSIZE; i++)
{
    // add one to odd values
    if (array[i] % 2 == 1)
    {
        array[i]++;
    }
}
```

References

- <https://www.csee.umbc.edu/courses/undergraduate/341/fall16/projects/coding-standards.shtml>
- https://en.wikipedia.org/wiki/Indentation_style
- <http://www.cs.nott.ac.uk/~pszcah/G53QAT/Presentations09/jjb07u/QAT09Presentations-jjb07u.ppt>

Software Engineering, Professional Ethics

Outline

- Ethics
- Ethical guidelines for computer professionals
- Codes of ethics for computing professionals
- Guidelines for software developers and decision makers
- The SE Code
- The ACM Code
- Computing Curricula ACM/IEEE

Ethical Guidelines for Computer Professionals

- We have responsibilities not only to our customers but also to the general public.
- Responsibilities including minimizing risks to:
 - ❑ Privacy
 - ❑ Security of data
 - ❑ Safety
 - ❑ Reliability
 - ❑ Ease of use
- Must exercise good practices to reduce likelihood of problems, including maintaining professional competency.

Codes of Ethics for Computing Professionals

- Software Engineering Code of Ethics and Professional Practice
 - ❑ Adopted by Association of Computing Machinery and IEEE Computing Society
 - ❑ <https://www.acm.org/about-acm/code-of-ethics>
 - ❑ Software Engineering Code of Ethics and Professional Practice (Version 5.2) as recommended by the ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices and jointly approved by the ACM and the IEEE-CS as the standard for teaching and practicing software engineering.

Guidelines for Software Developers and decision makers

- Include users in the design and testing stages – makes systems safe and useful
- Be careful when planning and scheduling a project, writing bids or contracts
- Design for real users (make it crash resistant)
- Don't assume existing software is safe
- Be open and honest about capabilities and limitations of software
- Require a convincing case for safety
 - A disastrous attitude: Challenger engineers had to prove “beyond a shadow of a doubt that it was not safe to [launch].”

The Software Engineering Code

8 principles that express responsibilities to:

1. The public
2. The client and employer
3. The product
4. Professional judgment
5. Management
6. The profession
7. Colleagues
8. Self

1. The public

1. Accept full responsibility for their own work.
2. Moderate the interests of the software engineer, the employer, the client and the users with the public good.
3. Approve software only if they have a well-founded belief that it is safe, meets specifications, passes appropriate tests, and does not diminish quality of life, diminish privacy or harm the environment. The ultimate effect of the work should be to the public good.
4. Disclose to appropriate persons or authorities any actual or potential danger to the user, the public, or the environment, that they reasonably believe to be associated with software or related documents.

1. The public (cont)

5. Cooperate in efforts to address matters of grave public concern caused by software, its installation, maintenance, support or documentation.
6. Be fair and avoid deception in all statements, particularly public ones, concerning software or related documents, methods and tools.
7. Consider issues of physical disabilities, allocation of resources, economic disadvantage and other factors that can diminish access to the benefits of software.
8. Be encouraged to volunteer professional skills to good causes and contribute to public education concerning the discipline.

2. The client and employer

1. Provide service in their areas of competence, being honest and forthright about any limitations of their experience and education.
2. Not knowingly use software that is obtained or retained either illegally or unethically.
3. Use the property of a client or employer only in ways properly authorized, and with the client's or employer's knowledge and consent.
4. Ensure that any document upon which they rely has been approved, when required, by someone authorized to approve it.
5. Keep private any confidential information gained in their professional work, where such confidentiality is consistent with the public interest and consistent with the law.

2. The client and employer (cont)

6. Identify, document, collect evidence and report to the client or the employer promptly if, in their opinion, a project is likely to fail, to prove too expensive, to violate intellectual property law, or otherwise to be problematic.
7. Identify, document, and report significant issues of social concern, of which they are aware, in software or related documents, to the employer or the client.
8. Accept no outside work detrimental to the work they perform for their primary employer.
9. Promote no interest adverse to their employer or client, unless a higher ethical concern is being compromised; in that case, inform the employer or another appropriate authority of the ethical concern.

3. The product

1. Strive for high quality, acceptable cost and a reasonable schedule, ensuring significant tradeoffs are clear to and accepted by the employer and the client, and are available for consideration by the user and the public.
2. Ensure proper and achievable goals and objectives for any project on which they work or propose.
3. Identify, define and address ethical, economic, cultural, legal and environmental issues related to work projects.
4. Ensure that they are qualified for any project on which they work or propose to work by an appropriate combination of education and training, and experience.
5. Ensure an appropriate method is used for any project on which they work or propose to work.

3. The product (cont)

6. Work to follow professional standards, when available, that are most appropriate for the task at hand, departing from these only when ethically or technically justified.
7. Strive to fully understand the specifications for software on which they work.
8. Ensure that specifications for software on which they work have been well documented, satisfy the users' requirements and have the appropriate approvals.
9. Ensure realistic quantitative estimates of cost, scheduling, personnel, quality and outcomes on any project on which they work or propose to work and provide an uncertainty assessment of these estimates.
10. Ensure adequate testing, debugging, and review of software and related documents on which they work.

3. The product (cont)

11. Ensure adequate documentation, including significant problems discovered and solutions adopted, for any project on which they work.
12. Work to develop software and related documents that respect the privacy of those who will be affected by that software.
13. Be careful to use only accurate data derived by ethical and lawful means, and use it only in ways properly authorized.
14. Maintain the integrity of data, being sensitive to outdated or flawed occurrences.
15. Treat all forms of software maintenance with the same professionalism as new development.

4. Professional judgment

1. Temper all technical judgments by the need to support and maintain human values.
2. Only endorse documents either prepared under their supervision or within their areas of competence and with which they are in agreement.
3. Maintain professional objectivity with respect to any software or related documents they are asked to evaluate.
4. Not engage in deceptive financial practices such as bribery, double billing, or other improper financial practices.
5. Disclose to all concerned parties those conflicts of interest that cannot reasonably be avoided or escaped.
6. Refuse to participate, as members or advisors, in a private, governmental or professional body concerned with software related issues, in which they, their employers or their clients have undisclosed potential conflicts of interest.

5. Management

1. Ensure good management for any project on which they work, including effective procedures for promotion of quality and reduction of risk.
2. Ensure that software engineers are informed of standards before being held to them.
3. Ensure that software engineers know the employer's policies and procedures for protecting passwords, files and information that is confidential to the employer or confidential to others.
4. Assign work only after taking into account appropriate contributions of education and experience tempered with a desire to further that education and experience.
5. Ensure realistic quantitative estimates of cost, scheduling, personnel, quality and outcomes on any project on which they work or propose to work, and provide an uncertainty assessment of these estimates.

5. Management (cont)

6. Attract potential software engineers only by full and accurate description of the conditions of employment.
7. Offer fair and just remuneration.
8. Not unjustly prevent someone from taking a position for which that person is suitably qualified.
9. Ensure that there is a fair agreement concerning ownership of any software, processes, research, writing, or other intellectual property to which a software engineer has contributed.
10. Provide for due process in hearing charges of violation of an employer's policy or of this Code.
11. Not ask a software engineer to do anything inconsistent with this Code.
12. Not punish anyone for expressing ethical concerns about a project.

6. The profession

1. Help develop an organizational environment favorable to acting ethically.
2. Promote public knowledge of software engineering.
3. Extend software engineering knowledge by appropriate participation in professional organizations, meetings and publications.
4. Support, as members of a profession, other software engineers striving to follow this Code.
5. Not promote their own interest at the expense of the profession, client or employer.
6. Obey all laws governing their work, unless, in exceptional circumstances, such compliance is inconsistent with the public interest.

6. The profession (cont)

7. Be accurate in stating the characteristics of software on which they work, avoiding not only false claims but also claims that might reasonably be supposed to be speculative, vacuous, deceptive, misleading, or doubtful.
8. Take responsibility for detecting, correcting, and reporting errors in software and associated documents on which they work.
9. Ensure that clients, employers, and supervisors know of the software engineer's commitment to this Code of ethics, and the subsequent ramifications of such commitment.
10. Avoid associations with businesses and organizations which are in conflict with this code.

6. The profession (cont)

11. Recognize that violations of this Code are inconsistent with being a professional software engineer.
12. Express concerns to the people involved when significant violations of this Code are detected unless this is impossible, counter-productive, or dangerous.
13. Report significant violations of this Code to appropriate authorities when it is clear that consultation with people involved in these significant violations is impossible, counter-productive or dangerous.

7. Colleagues

1. Encourage colleagues to adhere to this Code.
2. Assist colleagues in professional development.
3. Credit fully the work of others and refrain from taking undue credit.
4. Review the work of others in an objective, candid, and properly-documented way.
5. Give a fair hearing to the opinions, concerns, or complaints of a colleague.

7. Colleagues (cont)

6. Assist colleagues in being fully aware of current standard work practices including policies and procedures for protecting passwords, files and other confidential information, and security measures in general.
7. Not unfairly intervene in the career of any colleague; however, concern for the employer, the client or public interest may compel software engineers, in good faith, to question the competence of a colleague.
8. In situations outside of their own areas of competence, call upon the opinions of other professionals who have competence in that area.

8. Self

1. Further their knowledge of developments in the analysis, specification, design, development, maintenance and testing of software and related documents, together with the management of the development process.
2. Improve their ability to create safe, reliable, and useful quality software at reasonable cost and within a reasonable time.
3. Improve their ability to produce accurate, informative, and well-written documentation.
4. Improve their understanding of the software and related documents on which they work and of the environment in which they will be used.
5. Improve their knowledge of relevant standards and the law governing the software and related documents on which they work.

8. Self (cont)

6. Improve their knowledge of this Code, its interpretation, and its application to their work.
7. Not give unfair treatment to anyone because of any irrelevant prejudices.
8. Not influence others to undertake any action that involves a breach of this Code.
9. Recognize that personal violations of this Code are inconsistent with being a professional software engineer.

ACM Code

- 24 statements of personal responsibility, including:
 - ❑ General moral imperatives
 - ❑ Professional responsibilities
 - ❑ Organizational leadership imperatives
 - ❑ Compliance with the Code

Computing Curricula ACM/IEEE

- Social context of computing
- Methods and tools of analysis of ethical argument
- Professional and ethical responsibilities
- Risks and liabilities of safety-critical systems
- Intellectual property
- Privacy and civil liberties
- Social implications of the Internet
- Computer crime
- Philosophical foundations of ethics

The Increasing General Public Awareness on Ethical Aspects of Technology

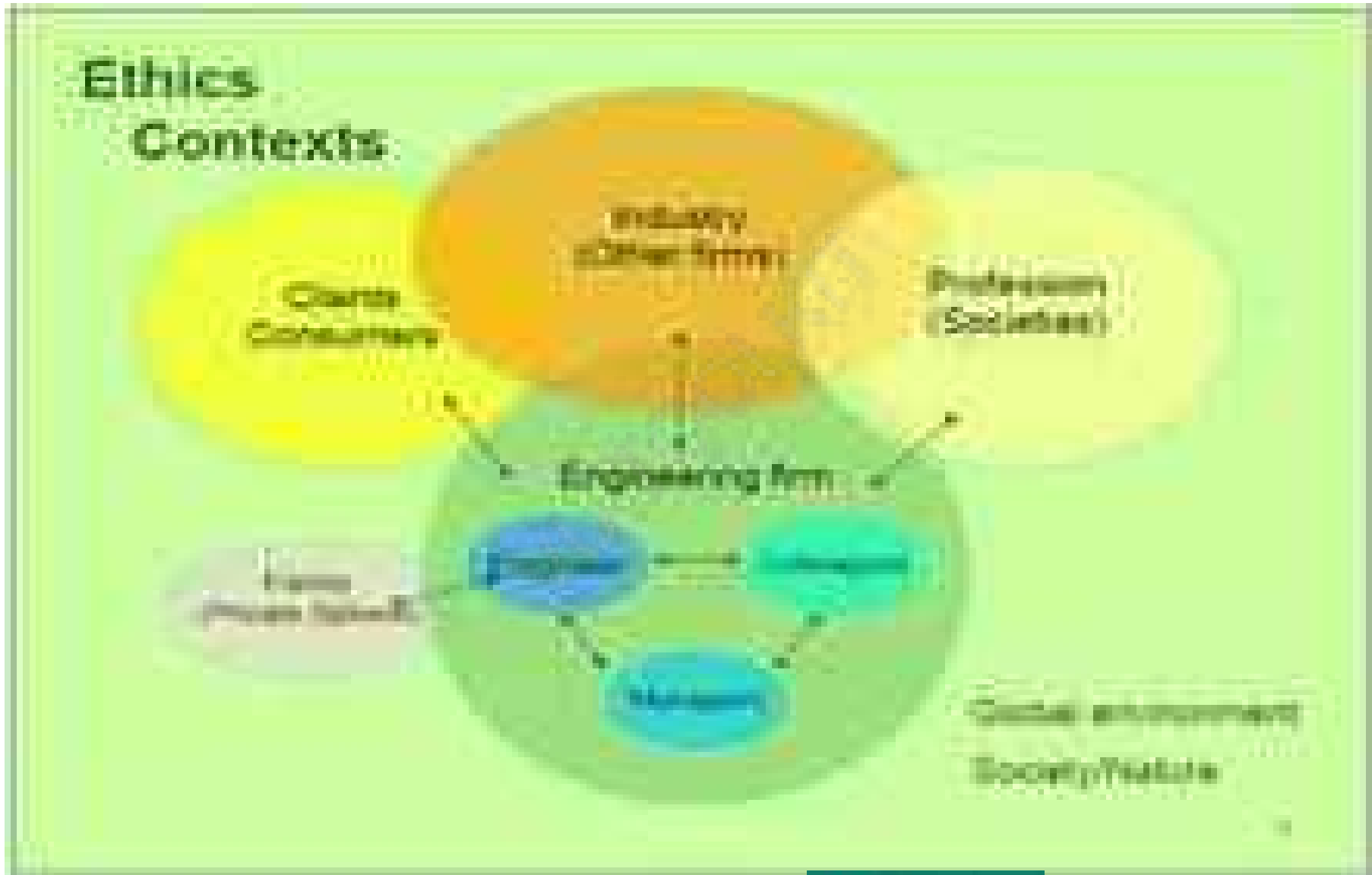
- The high level of media attention given to computer-related disasters in technical systems has increased interest in Computer Ethics:
 - ❑ The explosion of Ariane V in 1996
 - ❑ The Therac-25 computerized radiation machine overdoses

Why Ethics?

- *“There are few things wholly evil or wholly good. Almost everything...is an inseparable compound of the two, so that our best judgment of the preponderance between them is continually demanded.”*

Abraham Lincoln

Ethics Contexts



Engineering as Social Experimentation

- *“All products of technology present some potential dangers, and thus engineering is an inherently risky activity. In order to underscore this fact and help in exploring its ethical implications, we suggest that engineering should be viewed as an experimental process. It is not, of course, an experiment conducted solely in a laboratory under controlled conditions. Rather, it is an experiment on a social scale involving human subjects.”*

Ethics in Engineering, Martin MW and Schinzinger R, McGraw-Hill, 1996

Social Importance of Engineering

- Engineering has a direct and vital effect on the quality of life of people. 
- Accordingly, the services provided by engineers must be dedicated to the protection of the public safety, health and welfare.

Why is the Professional Ethics Important for Computer Scientists and Engineers?

- Because the Professional Ethics shall be a part of education *for every socially important profession, as one of essential constituents of the meaning of the term professionalism!*

Summary and Reading Suggestions

- Professional Ethics and Responsibilities
 - Sara Baase: “From A Gift of Fire”, Second Edition, 2003, Prentice Hall.
- American *ACM/IEEE Computing Curriculum*
 - <http://www.computer.org/education/cc2001/index.htm>

Software Construction and Development

Week-10

DevOps

- The DevOps is the combination of two words, one is **Development** and other is **Operations**. It is a culture to promote the development and operation process collectively.
- The DevOps will help you to learn DevOps basics and provide depth knowledge of various DevOps tools such as Git, Ansible, Docker, Puppet, Jenkins, Chef, Nagios, and Kubernetes.

Why DevOps?

- The operation and development team worked in complete isolation.
- After the design-build, the testing and deployment are performed respectively. That's why they consumed more time than actual build cycles.
- Without the use of DevOps, the team members are spending a large amount of time on designing, testing, and deploying instead of building the project.
- Manual code deployment leads to human errors in production.
- Coding and operation teams have their separate timelines and are not in synch, causing further delays.

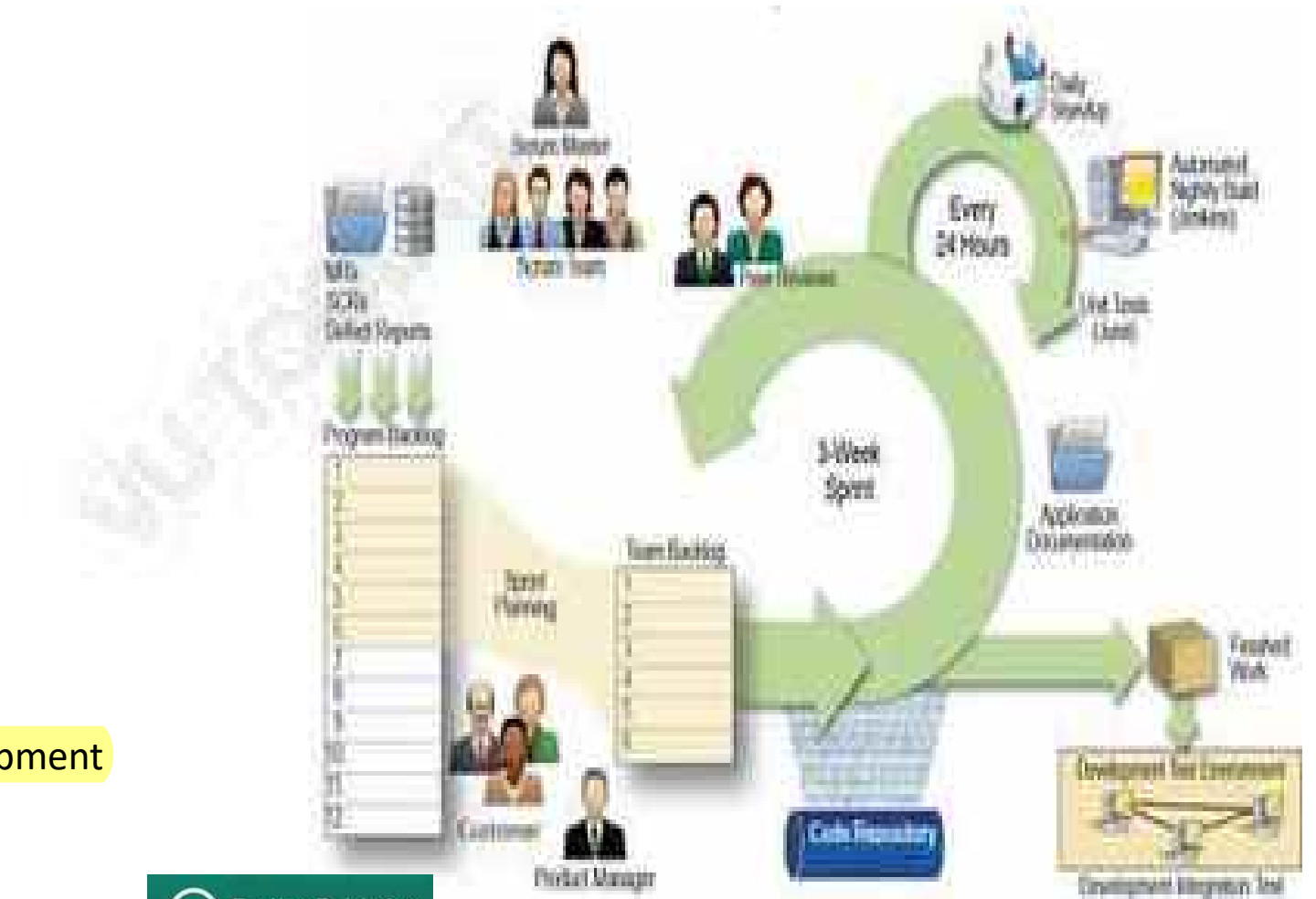
Water Fall Vs Agile

Waterfall Development Process

- Linear/Sequential
- Stages
- Advantages
- Disadvantages

Agile

- Iterative Process
 - Workflow
 - Advantages
 - Disadvantages
- Agile Vs Waterfall
 - Linear fashion of software development



Why DevOps?

- ◆ **Efficiency** - Faster time to market
- ◆ **Predictability** - Lower failure rate of new releases
- ◆ **Reproducibility** – Version everything
- ◆ **Maintainability** - Faster time to recovery in the event of a new release crashing or otherwise disabling the current system



Image from - dev2ops.org

Break down the wall between development and operations

DevOps Advantages

- ◆ DevOps is an excellent approach for quick development and deployment of applications.
- ◆ It responds faster to the market changes to improve business growth.
- ◆ DevOps escalate business profit by decreasing software delivery time and transportation costs.
- ◆ DevOps clears the descriptive process, which gives clarity on product development and delivery.
- ◆ It improves customer experience and satisfaction.
- ◆ DevOps simplifies collaboration and places all tools in the cloud for customers to access.
- ◆ DevOps means collective responsibility, which leads to better team engagement and productivity.

DevOps Disadvantages

- ◆ DevOps professional or expert's developers are less available.
- ◆ Developing with DevOps is so expensive.
- ◆ Adopting new DevOps technology into the industries is hard to manage in short time.
- ◆ Lack of DevOps knowledge can be a problem in the continuous integration of automation projects.

DevOps Architecture



DevOps Architecture

- **1) Build**

- Without DevOps, the cost of the consumption of the resources was evaluated based on the pre-defined individual usage with fixed hardware allocation. And with DevOps, the usage of cloud, sharing of resources comes into the picture, and the build is dependent upon the user's need, which is a mechanism to control the usage of resources or capacity.

- **2) Code**

- Many good practices such as Git enables the code to be used, which ensures writing the code for business, helps to track changes, and many others. The code can be appropriately arranged in files, folders, etc. And they can be reused.

- **3) Test**

- In the case of manual testing, it consumes more time in testing and moving the code to the output. The testing can be automated, which decreases the time for testing so that the time to deploy the code to production can be reduced as automating the running of the scripts will remove many manual steps.

- **4) Plan**

- DevOps use Agile methodology to plan the development. With the operations and development team in sync, it helps in organizing the work to plan accordingly to increase productivity.

DevOps Architecture

- **5) Monitor**

- Continuous monitoring is used to identify any risk of failure. Also, it helps in tracking the system accurately so that the health of the application can be checked. The monitoring becomes more comfortable with many third-party tools such as **Splunk**.

- **6) Deploy**

- Many systems can support the scheduler for automated deployment. The cloud management platform enables users to capture accurate insights and view the optimization scenario, analytics on trends by the deployment of dashboards.

- **7) Operate**

- DevOps changes the way traditional approach of developing and testing separately. The teams operate in a collaborative way where both the teams actively participate throughout the service lifecycle. The operation team interacts with developers, and they come up with a monitoring plan which serves the IT and business requirements.

- **8) Release**

- Deployment to an environment can be done by automation. But when the deployment is made to the production environment, it is done by manual triggering.

DevOps Lifecycle

- **1) Continuous Development**
- This phase involves the planning and coding of the software. The vision of the project is decided during the planning phase. And the developers begin developing the code for the application. There are no DevOps tools that are required for planning.
- **2) Continuous Integration**
- This stage is the heart of the entire DevOps lifecycle. It is a software development practice in which the developers require to commit changes to the source code more frequently on a daily or weekly basis. Building code is not only involved compilation, but it also includes unit testing, integration testing, code review, and packaging. Jenkins is a popular tool used in this phase.
- **3) Continuous Testing**
- This phase, where the developed software is continuously testing for bugs. For constant testing, automation testing tools such as TestNG, JUnit, Selenium, etc are used. These tools allow QAs to test multiple code-bases thoroughly in parallel to ensure that there is no flaw in the functionality. This entire testing phase can automate with the help of a Continuous Integration tool called Jenkins.



DevOps Lifecycle

- **4) Continuous Monitoring**

- Monitoring is a phase that involves all the operational factors of the entire DevOps process, where important information about the use of the software is recorded and carefully processed to find out trends and identify problem areas. It may occur in the form of documentation files or maybe produce large-scale data about the application parameters when it is in a continuous use position. The system errors such as server not reachable, low memory, etc are resolved in this phase. It maintains the security and availability of the service.

- **5) Continuous Feedback**

- The application development is consistently improved by analyzing the results from the operations of the software. This is carried out by placing the critical phase of constant feedback between the operations and the development of the next version of the current software application.

- **6) Continuous Deployment**

- In this phase, the code is deployed to the production servers. Also, it is essential to ensure that the code is correctly used on all the servers. The new code is deployed continuously, and configuration management tools play an essential role in executing tasks frequently and quickly. Here are some popular tools which are used in this phase, such as Chef, Puppet, Ansible, and SaltStack.

- **Continuous Operations**

- It is clear from the discussion that continuity is the critical factor in the DevOps in removing steps that often distract the development, take it longer to develop a better version of the product after several months. With DevOps, we can make a product more efficient and increase the overall



DevOps Principles

- **End to End Responsibility:**

DevOps team need to provide performance support until they become the end of life. It enhances the responsibility and the quality of the products engineered.

- **Continuous Improvement:**

- DevOps culture focuses on continuous improvement to minimize waste. It continuously speeds up the growth of products or services offered.

- **Automate Everything:**

- Automation is an essential principle of the DevOps process. This is for software development and also for the entire infrastructure landscape. 

- **Custom Centric Action:**

- DevOps team must take customer-centric for that they should continuously invest in products and services.

- **Monitor and test everything:**

- The DevOps team needs to have robust monitoring and testing procedures.

- **Work as one team:**

- In the DevOps culture role of the designers, developers, and testers are already defined. All they needed to do is work as one team with collaboration.



DevOps Tools

- **Puppet**
- Puppet is the most widely used DevOps tool. It allows the delivery and release of the technology changes quickly and frequently. It has features of versioning, automated testing, and continuous delivery.
- **Ansible**
- Ansible is a leading DevOps tool. Ansible is an open-source IT engine that automates application deployment, cloud provisioning, intra service orchestration, and other IT tools.
- **Docker**
- Docker is a high-end DevOps tool that allows building, ship, and run distributed applications on multiple systems.
- **Nagios**
- Nagios is one of the more useful tools for DevOps. It can determine the errors and rectify them with the help of network, infrastructure, server, and log monitoring systems.
- **Jenkins**
- Jenkins is a DevOps tool for monitoring the execution of repeated tasks. Jenkins is a software that allows continuous integration. Jenkins will be installed on a server where the central build will take place. It helps to integrate project changes more efficiently by finding the issues quickly.



Software Testing

- To understand the concept of software testing correctly, we need to understand a few related concepts.
- Verification and Validation



Verification and Validation

Verification

Does the product meet system specifications.

Have you built the product right?

Validation

Does the product meet user expectations.

Have you built the right product?

(Boehm)



Defect

- software defect is that phenomenon in which software deviates from its expected behavior. This is non-compliance from the expected behavior with respect to written specifications or the stakeholder needs.



Software Testing Objective

The correct approach to testing a scientific theory is not to try to verify it, but to seek to refute the theory – that is to prove that it has errors. (Popper 1965)

- The goal of testing is to expose latent defects in a software system before it is put to use.



Software Testing Objective

- A software tester tries to break the system.
- The objective is to show the presence of a defect.
- Testing cannot show the absence of a defect. It only increases your confidence level in the software.



Limitation of Testing

- A function to compare two strings of characters stored in an array for equality.

```
bool isStringsEqual(char a[], char b[]);
```



Limitation of Testing

a	b	Expected Result
"cat"	"dog"	false
"cat"	"cat"	true
"hen"	"hen"	true
"hen"	"heN"	false
"cat"	"CAT"	false
"cat"	"ball"	false
"cat"	"cat"	false
"HEN"	"hen"	false
"rat"	"door"	false
"cat"	"cat"	true



Code

```
bool IsStringsEqual(char a[], char b[])
{
    bool result;
    if (strlen(a) != strlen(b))
        result = false;
    else {
        for (int i = 0; i < strlen(a); i++)
            if (a[i] == b[i]) result = true;
            else result = false;
    }
    return result;
}
```



Limitation of Testing

- In order to prove that a formula or hypothesis is incorrect you have to show only one example.
- To prove that it is correct any number of examples are insufficient, You have to give a formal proof of its correctness.



Limitation of Testing

- These examples only help you in coming-up with a hypothesis but they are not proofs by themselves.
- They only enhance your comfort level in that particular hypothesis (i.e. in your piece of software).

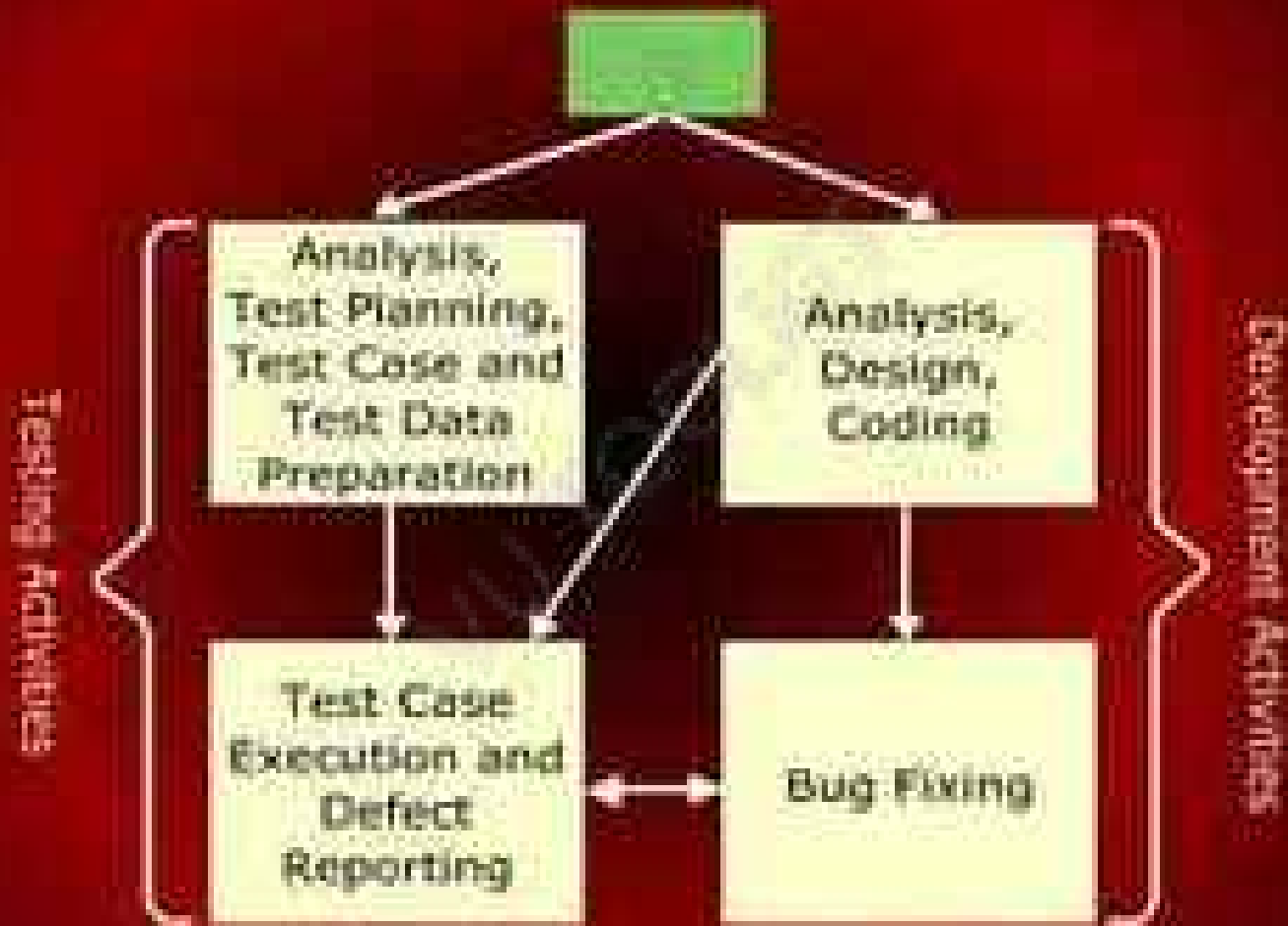


Test Cases and Test Data

- Input and output specification plus a statement of the function under test.
- Steps to perform the function
Expected results that the software application produces



Testing vs Development



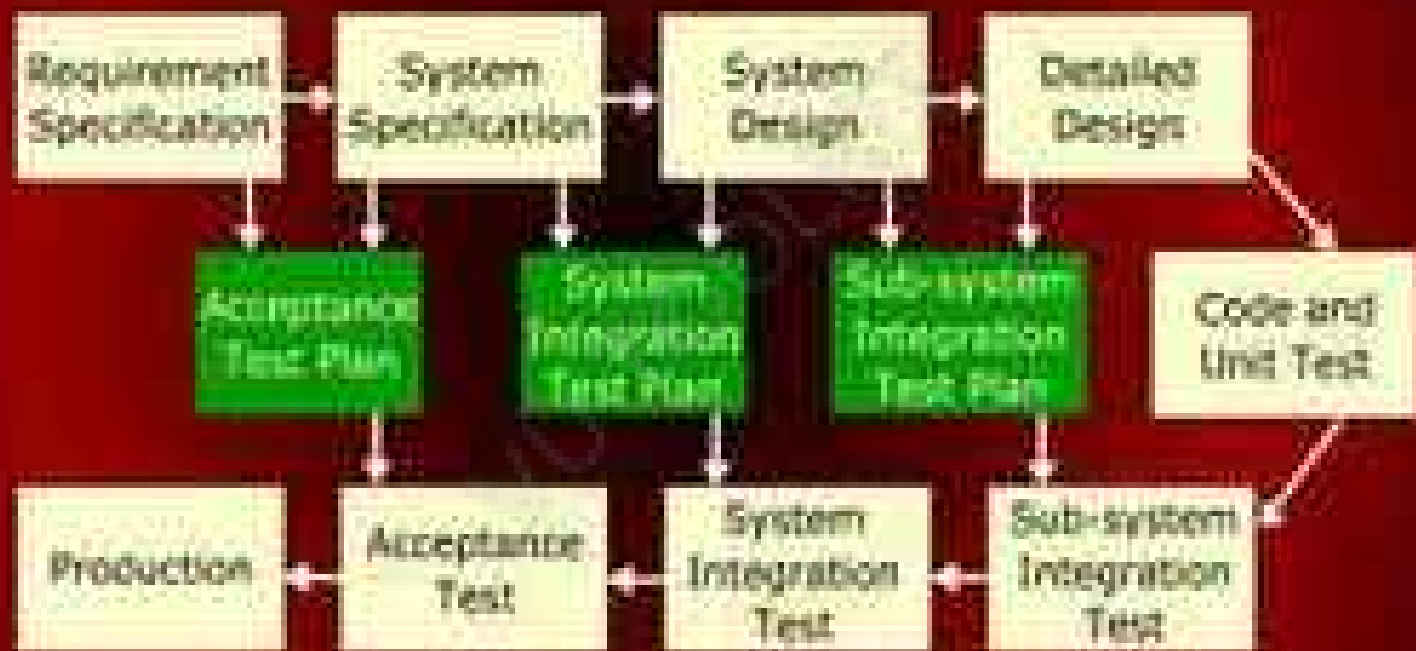
Developer as the Tester

Development Testing

- Development is a creative activity.
- The objective of development is to show that the program works.
- Testing is a destructive activity.
- The objective of testing is to show that the program does not work.



Testing and Software Phases



Testing Types

Structural or White Box Testing

- Analyses the internal structure of the program.
- Test cases are written to test these structures.



Guidelines for Finding Equivalence Classes

- Organize your equivalence classes.
- Determine groups of normal courses.
- Determine boundary conditions.
- Do not forget invalid inputs.



Example – isStringsEqual

- One string is empty.
 - First is NULL.
 - Second is NULL.



Example – isStringsEqual

2. Unequal Strings

- Two unequal strings of arbitrary length.
 - Two unequal strings of same length.
 - Two unequal strings of different length.
- Check for case sensitivity.
 - Same strings with different characters capitalized.



Example – isStringsEqual

- One string is empty.
 - First is NULL.
 - Second is NULL.



Test Cases - Equal Strings

- all lower case	"cat" "cat"	yes
- all upper case	"CAT" "CAT"	yes
- mixed case	"Cat" "Cat"	yes
- numeric values	"123" "123"	yes
- two strings with blanks only	" " " "	yes
- character and numeric mixed	"Cat1" "Cat1"	yes
- strings with special characters	"Cat#1" "Cat#1"	yes
- two NULL strings	"", ""	yes

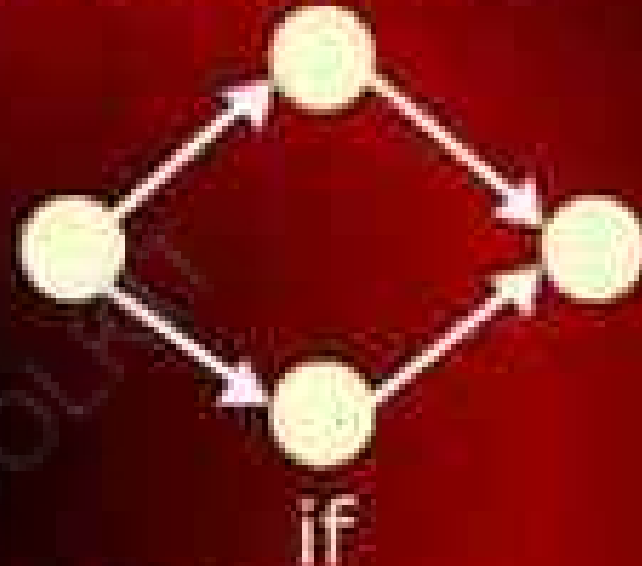
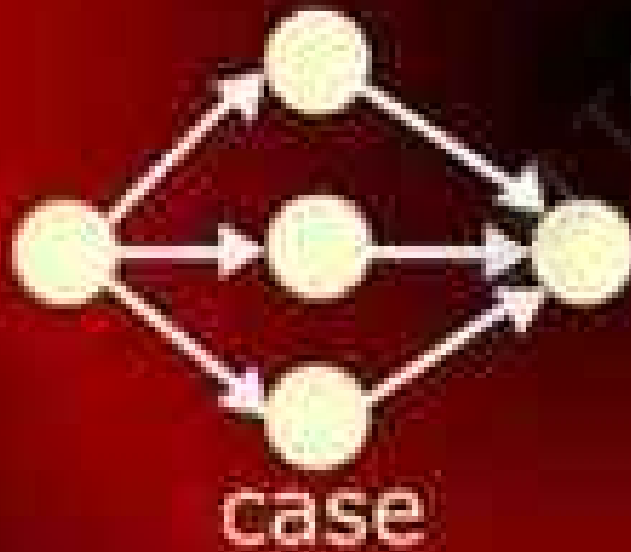


Test Cases - Unequal Strings

- two strings of different length	"cat" "mouse"	false
- two strings of same length	"cat" "dog"	false
- same strings with different characters capitalized	"Cat" "caT"	false
- first is NULL	NULL "cat"	false
- second is NULL	"cat" NULL	false



Flow Graph Notation



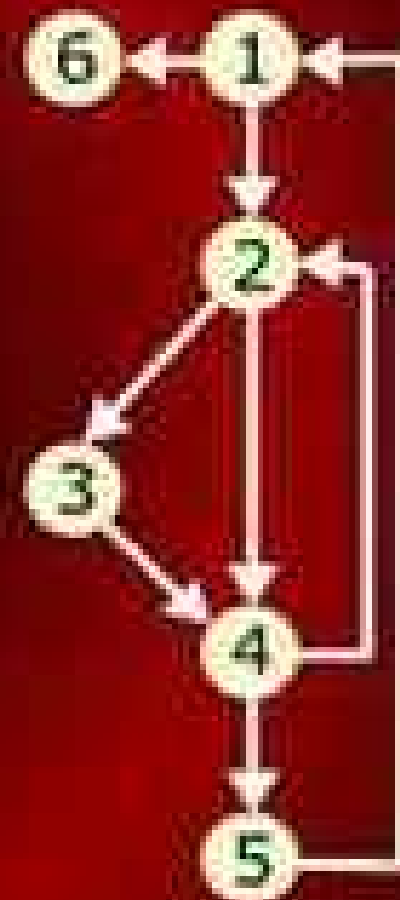
Flow Graph for a Sort Procedure

```
sorted = false;
while (!sorted) //1
{
    sorted = true;
    for (i=0; i < N-1; i++) //2
    {
        if (a[i] > a[i+1]) {
            swap(a[i], a[i+1]); //3
            sorted = false;
        } //4
    } //5
} //6
```



Flow Graph for a Sort Procedure

```
sorted = false;
while (!sorted) //1
{
    sorted = true;
    for (i=0; i < N-1; i++) //2
    {
        if (a[i] > a[i+1]) {
            swap(a[i], a[i+1]); //3
            sorted = false;
        } //4
    } //5
} //6
```



White Box Testing



White Box Testing

```
if (a == b)    //1  
    c = d;    //2  
a++;          //3
```



Statement Coverage
(One test case)

a=1, b=1



White Box Testing

```
if (a == b)    //1  
    c = d;    //2  
a++;          //3
```



Branch Coverage
(Two test cases)

a=1, b=1

a=1, b=2



White Box Testing

```
if (a == b)    //1  
    c = d;    //2  
a++;          //3
```

Path Coverage
(Two test cases)

a=1, b=1

a=1, b=2



Path Coverage

```
for (i = 0; i < N; i++)           //1
{
    if (condition1)
        // do something           //2
    else
        // do something           //3
        // do something           //4
}                                  //5
```



Path Coverage

```
for (i = 0; i < N; i++) //1
{
    if (condition1)
        // do something //2
    else
        // do something //3
        // do something //4
} //5
```

When $N = 0$
1 - 5



Path Coverage

```
for (i = 0; i < N; i++) //1
{
    if (condition1) //2
        // do something
    else //3
        // do something
    // do something //4
} //5
```

When N = 1

1 - 2 - 4 - 1 - 5

1 - 3 - 4 - 1 - 5



Path Coverage

```
for (i = 0; i < N; i++) //1
{
    if (condition1)
        // do something //2
    else
        // do something //3
        // do something //4
}
//5
```

When N = 2

1 - 2 - 4 - 1 - 2 - 4 - 1 - 5
1 - 2 - 4 - 1 - 3 - 4 - 1 - 5
1 - 3 - 4 - 1 - 2 - 4 - 1 - 5
1 - 3 - 4 - 1 - 3 - 4 - 1 - 5



Path Coverage

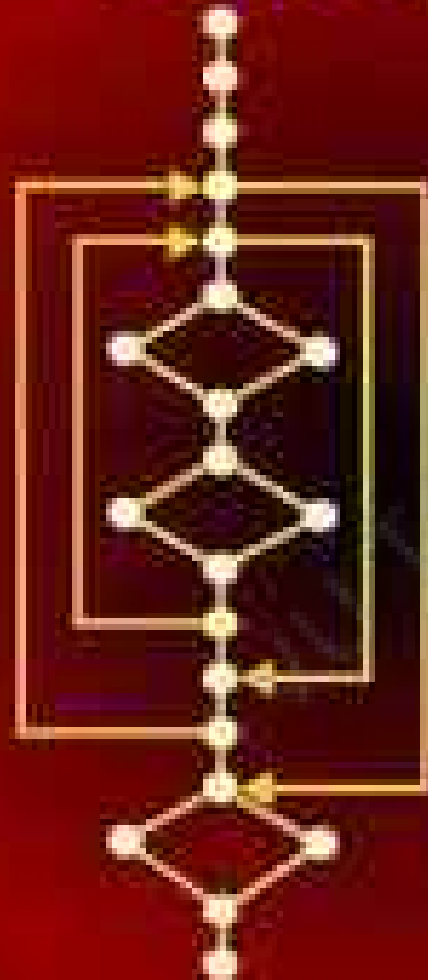
```
for (i = 0; i < N; i++) //1
{
    if (condition1)
        // do something //2
    else
        // do something //3
        // do something //4
} //5
```

When $N = 20$

Total number of paths are
more than 1 million.



Path Coverage



- 1852 different paths



Cyclomatic Complexity

Independent Path

Any path through the program (from start to end) that introduces at least one new set of processing statements or a new condition.

- It covers statements and all the branches.



Cyclomatic Complexity

- It is a quantitative measure of the logical complexity of a program.
- It defines the number of independent paths in the basis set of a program.



Cyclomatic Complexity

Cyclomatic Complexity, $C(G)$, for a flow graph G is defined as:

$$C(G) = E - N + 2$$

where E is the number of edges and N is the number of nodes in the flow graph G .



Cyclomatic Complexity

- It provides an upper bound for the number of tests that must be conducted to ensure that all statements and branches have been executed at least once.



Cyclomatic Complexity (Example)

```
sorted = false;
while (!sorted) { //1
    sorted = true;
    for (i=0; i < N-1; i++) { //2
        if (a[i] > a[i+1]) { //3
            swap(a[i], a[i+1]); //4
            sorted = false; //5
        }
    }
} //6
```



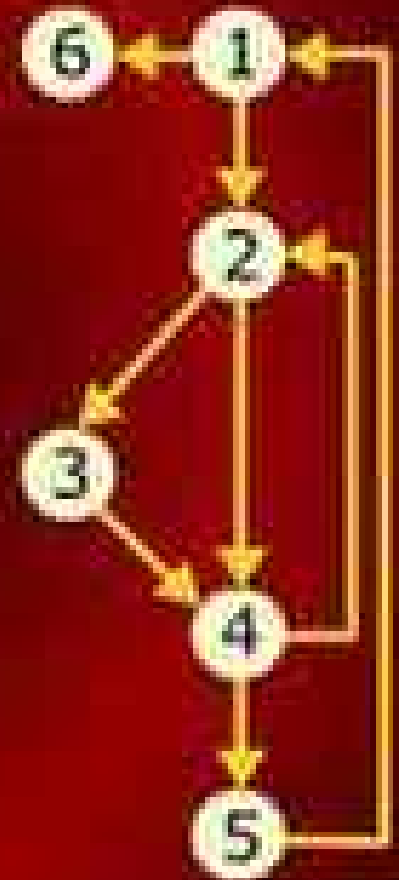
Cyclomatic Complexity (Example)

```
sorted = False;
while (sorted) {
    sorted = true;
    for (i=0; i < A.length; i++) {
        if (A[i] > A[i+1]) {
            swap(A[i], A[i+1]);
            sorted = false;
        }
    }
}
```

Number of edges = 8

Number of nodes = 6

$$C(G) = 8 - 6 + 2 = 4$$



Cyclomatic Complexity (Example)

```

sorted = false;
while (sorted) {
    sorted = true;
    for (i=0; i < A.length; i++) {
        if (A[i] > A[i+1]) {
            swap(A[i], A[i+1]);
            sorted = false;
        }
    }
}
    
```

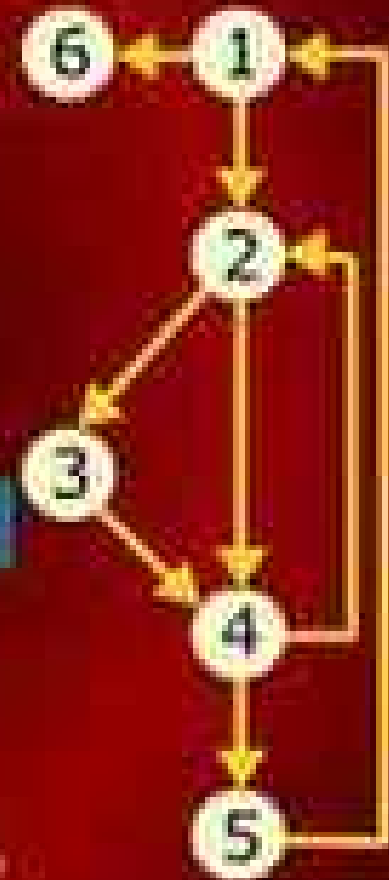
Independent Paths

1 - 6

1 - 2 - 3 - 4 - 5 - 1 - 6

1 - 2 - 4 - 5 - 1 - 6

1 - 2 - 4 - 2 - 3 - 4 - 5 - 1 - 6



Infeasible Paths

```
if (a == b)           //1
    c = c-1;          //2
if (a != b)           //3
    c = c+1;          //4
                      //5
```

Infeasible Paths

1-2-3-4-5

1-3-5



Infeasible Paths

A path through a program which is never traversed for any input data.

VU TOOLKIT



Path Coverage

```
for (i = 0; i < N; i++) //1
{
    if (condition1) //2
        // do something
    else //3
        // do something
    // do something //4
} //5
```

When N = 1

1 - 2 - 4 - 1 - 5

1 - 3 - 4 - 1 - 5



Path Coverage

```
for (i = 0; i < N; i++) //1
{
    if (condition1) //2
        // do something
    else //3
        // do something
    // do something //4
} //5
```

Total number of possible paths are 2^N .



Developer as the Tester

Development Testing

- Development is a creative activity.
- The objective of development is to show that the program works.
- Testing is a destructive activity.
- The objective of testing is to show that the program does not work.



Version control with Git

week 12

Problems Working Alone

- Ever done one of the following?

- Had code that worked, made a bunch of changes and saved it, which broke the code, and now you just want the working version back...
- Accidentally deleted a critical file, hundreds of lines of code gone...
- Somehow messed up the structure/contents of your code base, and want to just “undo” the crazy action you just did
- Hard drive crash!!!! Everything’s gone, the day before deadline.

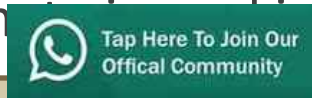
- Possible options:

- Save as (MyClass-v1.java)
 - Ugh. Just ugh. And now a single line change results in duplicating the entire file...

Problems Working in teams

- Whose computer stores the "official" copy of the project?
 - Can we store the project files in a neutral "official" location?
- Will we be able to read/write each other's changes?
 - Do we have the right file permissions?
 - Lets just email changed files back and forth! Yay!
- What happens if we both try to edit the same file?
 - Bill just overwrote a file I worked on for 6 hours!
- What happens if we make a mistake and corrupt an important file?
 - Is there a way to keep backups of our project files?

■ How do I know what code each team member is working on?



Solution: Version Control

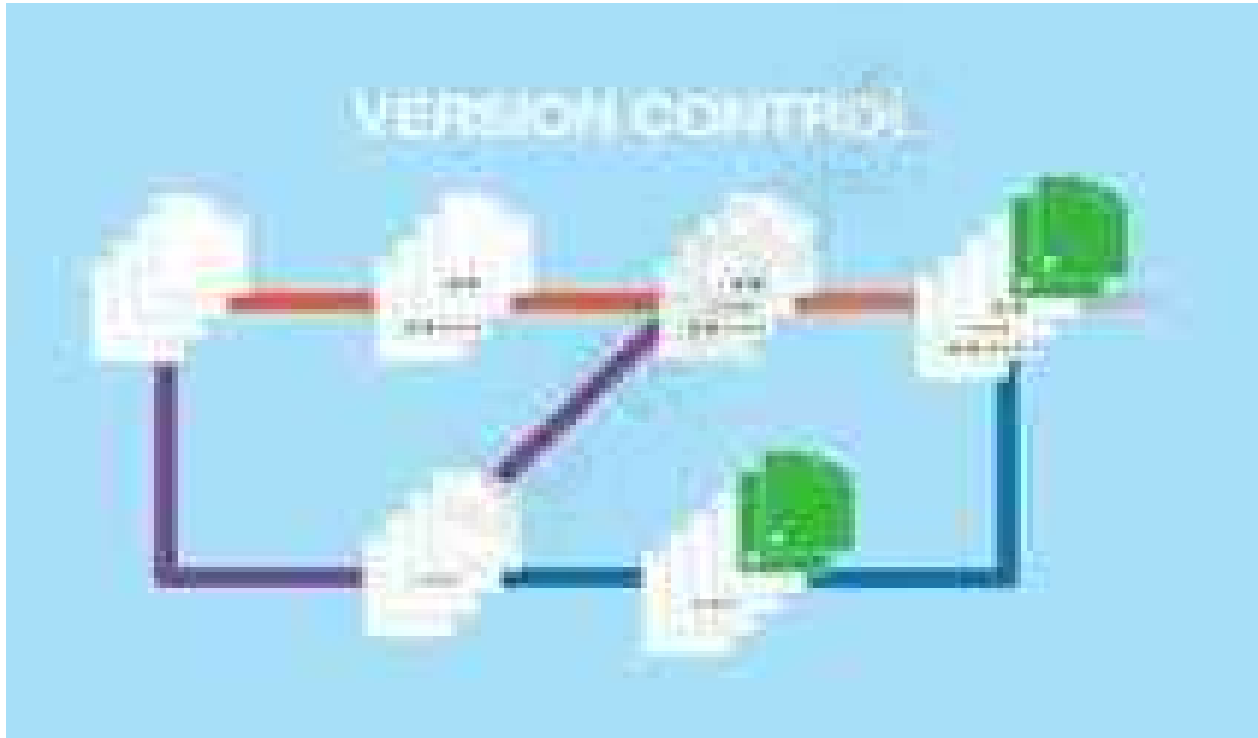
- **version control system:** Software that tracks and manages changes to a set of files and resources.
- You use version control all the time
 - Built into word processors/spreadsheets/presentation software
 - The magical “undo” button takes you back to “the version before my last action”
 - Wiki's
 - Wiki's are all about version control, managing updates, and allowing rollbacks to previous versions

Software Version control

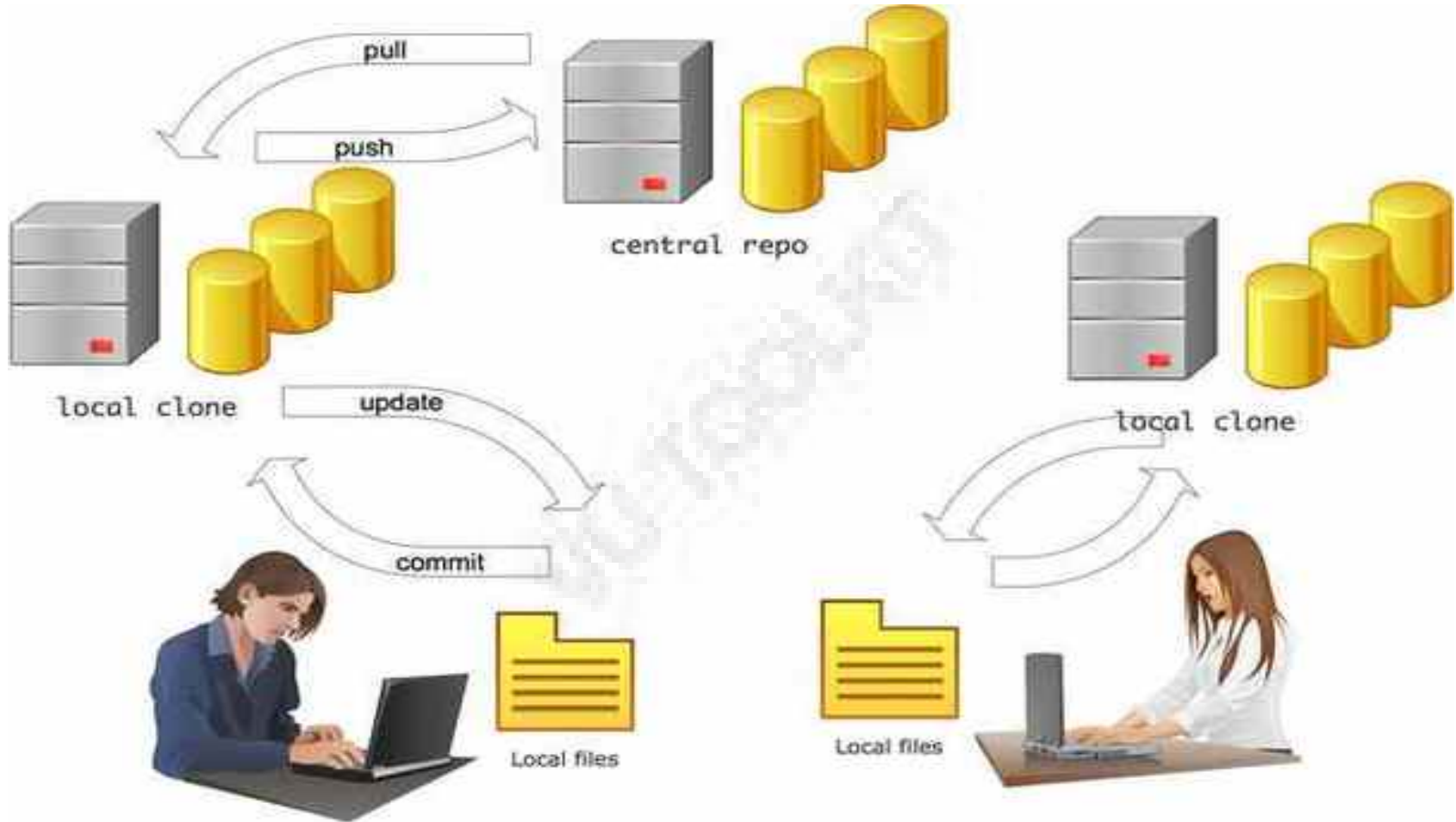
- Many version control systems are designed and used especially for software engineering projects
 - examples: CVS, Subversion (SVN), **Git**, Monotone, BitKeeper, Perforce
- helps teams to work together on code projects
 - a shared copy of all code files that all users can access
 - keeps current versions of all files, and backups of past versions
 - can see what files others have modified and view the changes
 - manages conflicts when multiple users modify the same file
- not particular to source code; can be used for papers, photos, etc.
 - but often works best with plain text/code files

version Control

- Version Control



Source Code Management



Version control tools



What is Git?

- Git is a popular version control system. It was created by Linus Torvalds in 2005, and has been maintained by Junio Hamano since then.

- **It is used for:**

- Tracking code changes
- Tracking who made changes
- Coding collaboration

- **What does Git do?**

- Manage projects with **Repositories**
- **Clone** a project to work on a local copy
- Control and track changes with **Staging** and **Committing**
- **Branch** and **Merge** to allow for work on different parts and versions of a project
- **Pull** the latest version of the project to a local copy
- **Push** local updates to the main project

Working with Git

- Initialize Git on a folder, making it a **Repository**
- Git now creates a hidden folder to keep track of changes in that folder
- When a file is changed, added or deleted, it is considered **modified**
- You select the modified files you want to **Stage**
- The **Staged** files are **Committed**, which prompts Git to store a **permanent** snapshot of the files
- Git allows you to see the full history of every commit.
- You can revert back to any previous commit.
- Git does not store a separate copy of every file in every commit, but keeps track of changes made in each commit!

Why Git?

Over 70% of developers use Git!

Developers can work together from anywhere in the world.

Developers can see the full history of the project.

Developers can revert to earlier versions of a project.

Image from - dev2ops.org

What is GitHub?

Git is not the same as GitHub.

GitHub makes tools that use Git.

GitHub is the largest host of source code in the world, and has been owned by Microsoft since 2018.

Git Install:

You can download Git for free from the following website:

<https://www.git-scm.com/>

Repositories

- **Repository (aka “repo”)**: a location storing a copy of all files.
 - you don't edit files directly in the repo;
 - you edit a local **working copy** or “working tree”
 - then you **commit** your edited files into the repo
- There may be only one repository that all users share (CVS, Subversion)
- Or each user could also have their own copy of the repository (Git, Mercurial)
- Files in your working directory must be **added** to the repo in order to be tracked.

What to put in a Repo?

- Everything needed to create your project:
 - Source code (Examples: .java, .c, .h, .cpp)
 - Build files (Makefile, build.xml)
 - Other resources needed to build your project: icons, text etc.
- Things generally NOT put in a repo (these can be easily re-created and just take up space):
 - Object files (.o)
 - Executables (.exe)

Repository Location

- Can create the repository anywhere

- Can be on the same computer that you're going to work on, which might be ok for a personal project where you just want rollback protection

- But, usually you want the repository to be robust:

- On a computer that's up and running 24/7
 - Everyone always has access to the project
- On a computer that has a redundant file system (ie RAID)
 - No more worries about that hard disk crash wiping away your project!

- Options:

- attu, CSE GitLab, GitHub (do NOT use GitHub for homework!!!)

Aside: So what is GitHub?

- [GitHub.com](https://github.com) is a site for online storage of Git repositories.
- Many open source projects use it, such as the [Linux kernel](https://www.kernel.org/).
- You can get free space for open source projects or you can pay for private projects.
- Do NOT use GitHub to store your homework!!

Question: Do I have to use GitHub to use Git?

Answer: No!

- you can use Git completely locally for your own purposes, or

Git Resources

- At the command line: (where <verb> = config, add, commit, etc.)

```
$ git help <verb>
```

```
$ git <verb> --help
```

```
$ man git-<verb>
```

- Free on-line book: <https://git-scm.com/book/en/v2>
- Git tutorial: <http://schacon.github.com/git/gittutorial.html>
- Reference page for Git: <http://gitref.org/index.html>
- Git website: <http://git-scm.com/>
- Git for Computer Scientists: <http://eagain.net/articles/git-for-computer-scientists/>

Centralized vs Distributed

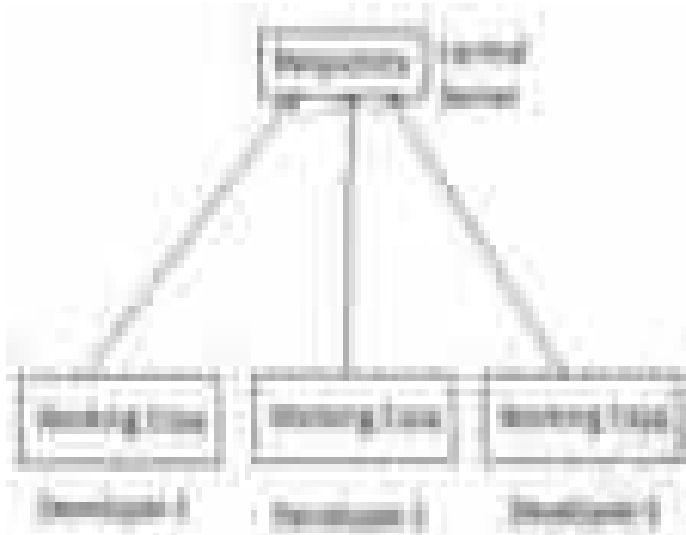


Fig. 20a

Centralized System Architecture

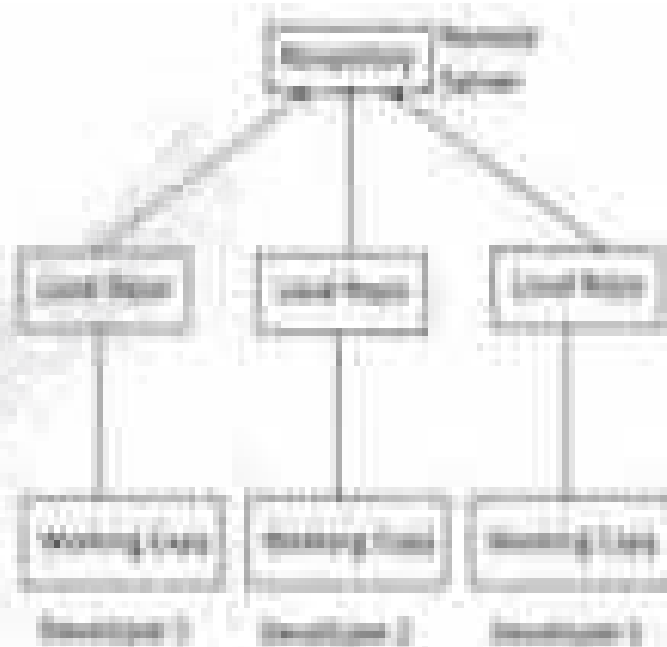


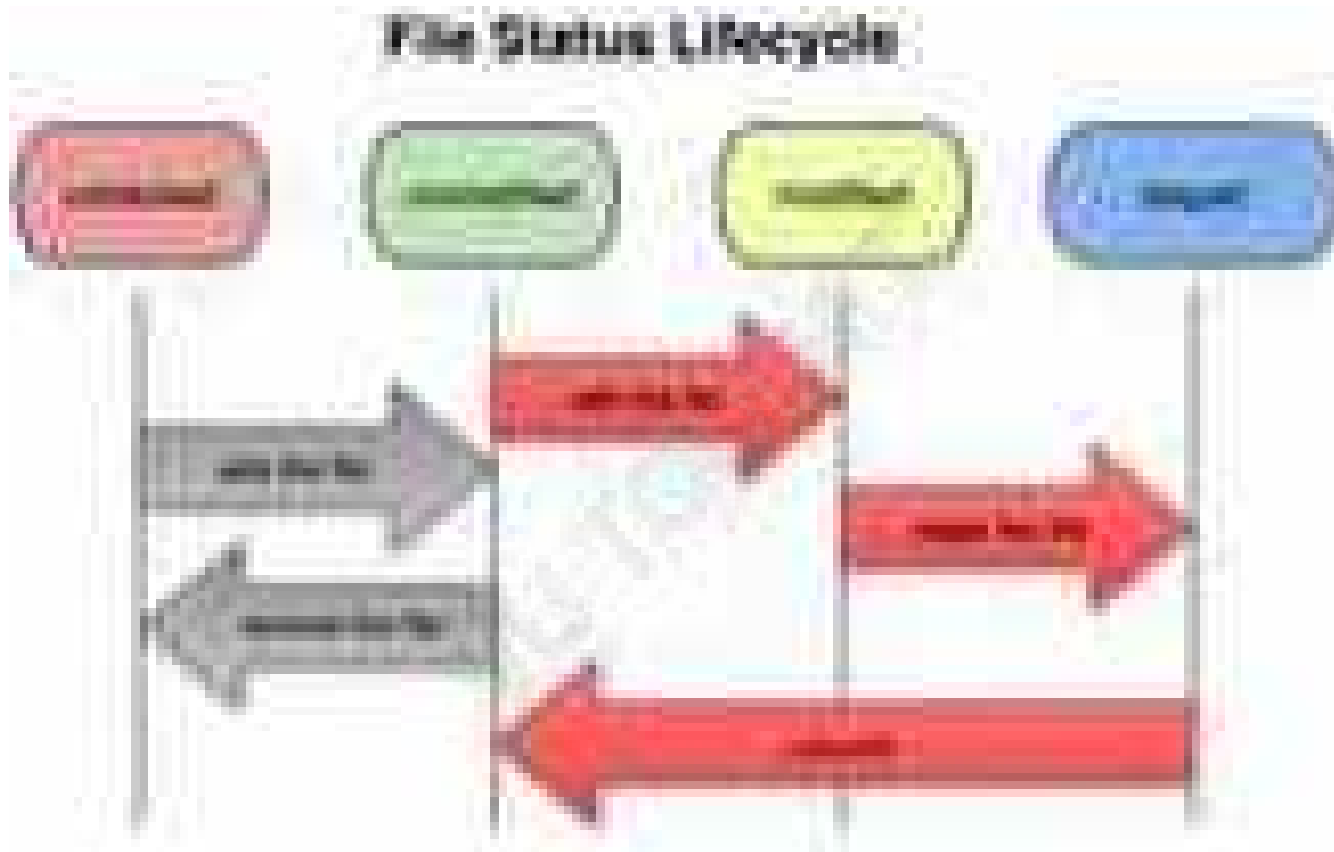
Fig. 20b

Distributed System Architecture

Git Features



Git file lifecycle



Git Installation



Basic Workflow

Basic Git workflow:

1. **Modify** files in your working directory.
2. **Stage** files, adding snapshots of them to your staging area.
3. Do a **commit**, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory (your local copy of the repo).

- **Notes:**

- If a particular version of a file is in the **git directory**, it's considered **committed**.
- If it's modified but has been added to the **staging area**, it is **staged**.
- If it was **changed** since it was checked out but has not been staged, it is **modified**.

Get ready to use Git!

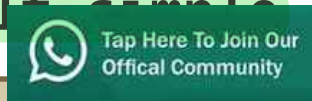
1. Set the name and email for Git to use when you commit:

```
$ git config --global user.name "Bugs Bunny"
```

```
$ git config --global user.email bugs@gmail.com
```

- You can call `git config --list` to verify these are set.
- These will be set globally for all Git projects you work with.
- You can also set variables on a project-only basis by not using the `--global` flag.
- You can also set the editor that is used for writing commit messages:
`$ git config --global core.editor emacs` (it is vim by default)
- The latest version of git will also prompt you that `push.default` is not set, you can make this warning go away with:

```
$ git config --global push.default simple
```



Create a local copy of a repo

2. Two common scenarios: (only do one of these)

a) To clone an already existing repo to your current directory:

```
$ git clone <url> [local dir name]
```

This will create a directory named *local dir name*, containing a working copy of the files from the repo, and a **.git** directory which you can ignore (used to hold the staging area and your actual repo)

Example: `git clone git@gitlab.cs.washington.edu:rea/superTest.git`

b) To create a Git repo in your current directory:

```
$ git init
```

This will create a **.git** directory in your current directory which you can ignore (used to hold the staging area and your actual repo).

Then you can commit files in your current directory into the repo:

```
$ git add file1.java
```

```
$ git commit -m "initial project"
```

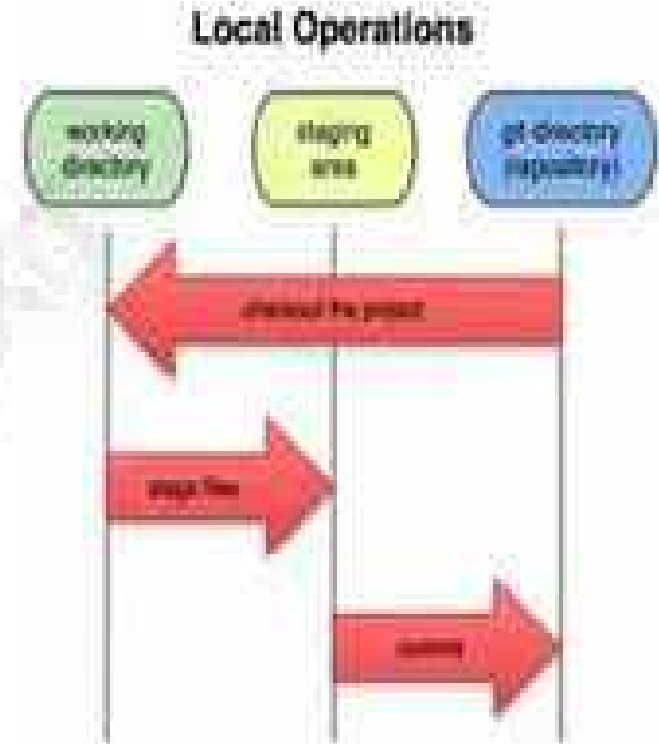


Git operations & Commands

- Download Git and Install
- Run Terminal
- GiT Bash Open (check Git Version using command `git--version` and press enter)
- Create Sample Project Folder using `mkdir` command
 - by default in C where windows resides
 - Path (C\Users\User name\folder name)
- Create a file in which we have to work using “`cd`” command
- Add webpage using Notepad++ editor
- we can create multiple files on the same location
- Now, we use GiT for these two files

Git operations & Commands

- Now, we use GiT for these two files “home.html and page.html” using GiT Initiate
- GiT init command (GiT named folder is created in working directory)
- Staging Area command
 - `git add home.html`
 - after changes in html file
 - run `git diff` command
- Now, we can track changes



Git operations & Commands

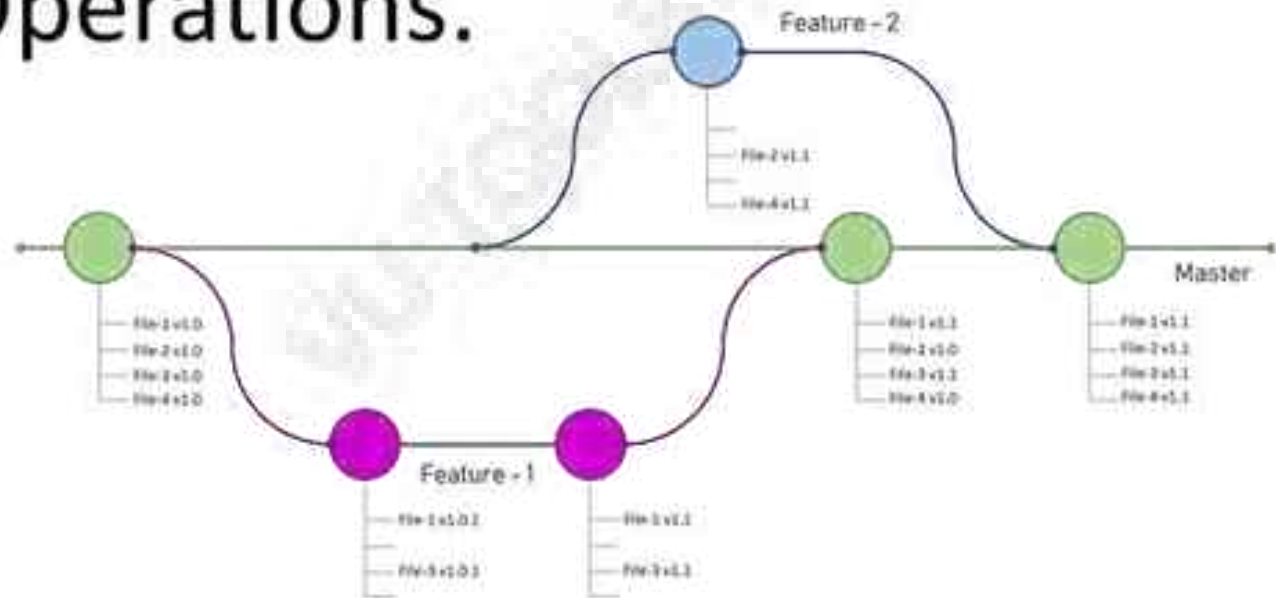
- “Git status” command will tell you the number of added files in staging area
- How to push and pull files on Local and Remote Directories from staging area?
 - git commit-a-m (push all files to local repository and an attached message what, where and when to commit changes in each file-track changes)
 - Then, push files to remote repository, for this sign up to GitHub
 - Create Repository with “New” option. <https://github.com/join>
 - New/name of repository/public or private/ Create (only one)
 - Use URL of Repository to push files from LR to RR.

Git operations & Commands

- Add Repository in Git using command (can add multiple..)
 - “git remote add <repository name> <repository url> ”
- example git remote add origin https://github.com....
- Now, push files from LR to RR, use the command
 - “git push origin master”
 - credentials for the first time
- Add a new branch using command
 - git branch fix
 - git branch command will show the total created branches
 - fix and master

Branching

GIT Branch and its Operations.



Branching

To create a branch called experimental:

- `$ git branch experimental`

To list all branches: (* shows which one you are currently on)

- `$ git branch`

To switch to the experimental branch:

- `$ git checkout experimental`

Later on, changes between the two branches differ, to merge changes from experimental into the master:

- `$ git checkout master`
- `$ git merge experimental`

Note: `git log --graph` can be useful for showing branches. 

Git commands

command	description
git clone <i>url [dir]</i>	copy a git repository so you can add to it
git add <i>files</i>	adds file contents to the staging area
git commit	records a snapshot of the staging area
git status	view the status of your files in the working directory and staging area
git diff	shows diff of what is staged and what is modified but unstaged
git help <i>[command]</i>	get help info about a particular command
git pull	fetch from a remote repo and try to merge into the current branch
git push	push your new branches and data to a remote repository

others: init, reset, branch, checkout, merge, log, tag

After editing a file...

```
$ emacs rea.txt
```

```
$ git status
```

```
# On branch master
```

```
# Changes not staged for commit:
```

```
# (use "git add <file>..." to update what will be committed)
```

```
# (use "git checkout -- <file>..." to discard changes in working directory)
```

```
#
```

```
#    modified:   rea.txt
```

```
#
```

```
no changes added to commit (use "git add" and/or "git commit -a")
```

```
$ git status -s
```

```
M rea.txt
```

← Note: M is in second column = "working tree"

```
$ git diff
```

← Shows modifications that have not been staged.

```
diff --git a/rea.txt b/rea.txt
```

```
index 66b293d..90b65fd 100644
```

```
--- a/rea.txt
```

```
+++ b/rea.txt
```

```
@@ -1,2 +1,4 @@
```

```
Here is rea's file.
```

```
+
```

```
+One new line added.
```

```
$ git diff --cached
```

← Shows nothing, no modification

```
$
```



After adding file to staging area...

```
$ git add rea.txt
```

```
$ git status
```

```
# On branch master
```

```
# Changes to be committed:
```

```
# (use "git reset HEAD <file>..." to unstage)
```

```
#
```

```
#    modified:   rea.txt
```

```
#
```

```
$ git status -s
```

```
M rea.txt
```

```
$ git diff
```

```
$ git diff --cached
```

```
diff --git a/rea.txt b/rea.txt
```

```
index 66b293d..90b65fd 100644
```

```
--- a/rea.txt
```

```
+++ b/rea.txt
```

```
@@ -1,2 +1,4 @@
```

```
Here is rea's file.
```

```
+
```

```
+One new line added.
```

← Note: M is in first column = "staging area"

← Note: Shows nothing, no modifications that have not been staged.

← Note: Shows staged modifications.

Pulling and Pushing

Good practice:

1. **Add** and **Commit** your changes to your local repo
 2. **Pull** from remote repo to get most recent changes (fix conflicts if necessary, add and commit them to your local repo)
 3. **Push** your changes to the remote repo
-

To fetch the most recent updates from the remote repo into your local repo, and put them into your working directory:

```
$ git pull origin master
```

To push your changes from your local repo to the remote repo:

```
$ git push origin master
```

Notes: **origin** = an alias for the URL you cloned from

master = the remote branch you are pulling from/pushing to,
(the local branch you are pulling to/pushing from is your current branch)

Software construction and development Practical - I

Week 13-Canban Practical

Create a new project in Asana

— — —



Add and Assign Tasks



Set Timelines for a Task

— — —



Project Manager Dashboard View



Developer View of Pending and Completed Tasks



Work Space and Team

— — —



Software construction and development Practical - I

Week 13-DevOps-Practical

DEVOPS

— — —



Software Development Practices

— — —

- *Programming Languages*
- *Version Control Systems*
- *Methodologies*
- *Software Development Life Cycle (SDLC)*

Operating System and Networking

— — —

Learn system administration concepts such as operating systems, networking, server management, security, and virtualization technologies.

CI/CD

— — —

Explore and master key DevOps practices like continuous integration and continuous delivery (CI/CD), including tools like Jenkins, Travis CI, and GitLab CI/CD.

Essential DevOps tools

— — —

Acquire proficiency in essential DevOps tools and technologies, such as configuration management (Ansible, Chef, Puppet), containerization and orchestration (Docker, Kubernetes), cloud computing platforms (AWS, Azure, GCP), infrastructure automation (Terraform, CloudFormation), and monitoring/logging (Prometheus, Grafana, ELK stack).

Software construction and development Practical - I

Week 13-Git Practical

GitLab

Home Projects Groups Settings

Create new project



Create blank project

Create a blank project to start your new work, and collaborate on code, among other things.



Create from template

Create a project pre-populated with the necessary files to get your desired results.



Import project

Register your code from an external source like GitHub, Bitbucket, or another instance of GitLab.



Run CI/CD for external repository

Connect your external repository to GitLab CI/CD.

New Project

Home > Projects > New project > Create blank project



Create blank project

Create a blank project to store your files, plan your work, and collaborate on tasks, among other things.

Project name

Must start with a lowercase letter, contain only letters, digits, and hyphens. Can also contain dots, spaces, dashes, or underscores.

Project ID

Must be unique across all projects under the same namespace. Create a ID.

Project tag

Watch deployment target (optional)

Visibility Level

☒ **Private**
Project access must be granted explicitly to each user. If this project is part of a group, access is granted to members of the group.

☐ **Public**
The project can be accessed without any authorization.

Project Configuration

☒ **Include telemetry with a SSOID**
Please refer to documentation about the project's telemetry. See also if you want to switch an existing dashboard.

☐ **Enable State Replication Network, using SSOID**
[Read the docs](#) to learn more about enabling state replication. [Learn more](#)

Git init

— — —





Git Push

— — —





Why?

- Why testing?
 - Improve software design
 - Make software easier to understand
 - Reduce debugging time
 - Catch integration errors
- In short, to Produce Better Code
- Preconditions
 - Working code
 - Good set of unit tests





What should be tested ?

- Test for boundary conditions
- Test for both success and failure
- Test for general functionality
- Etc..





When to start testing

Software quality and testing is a life-cycle process





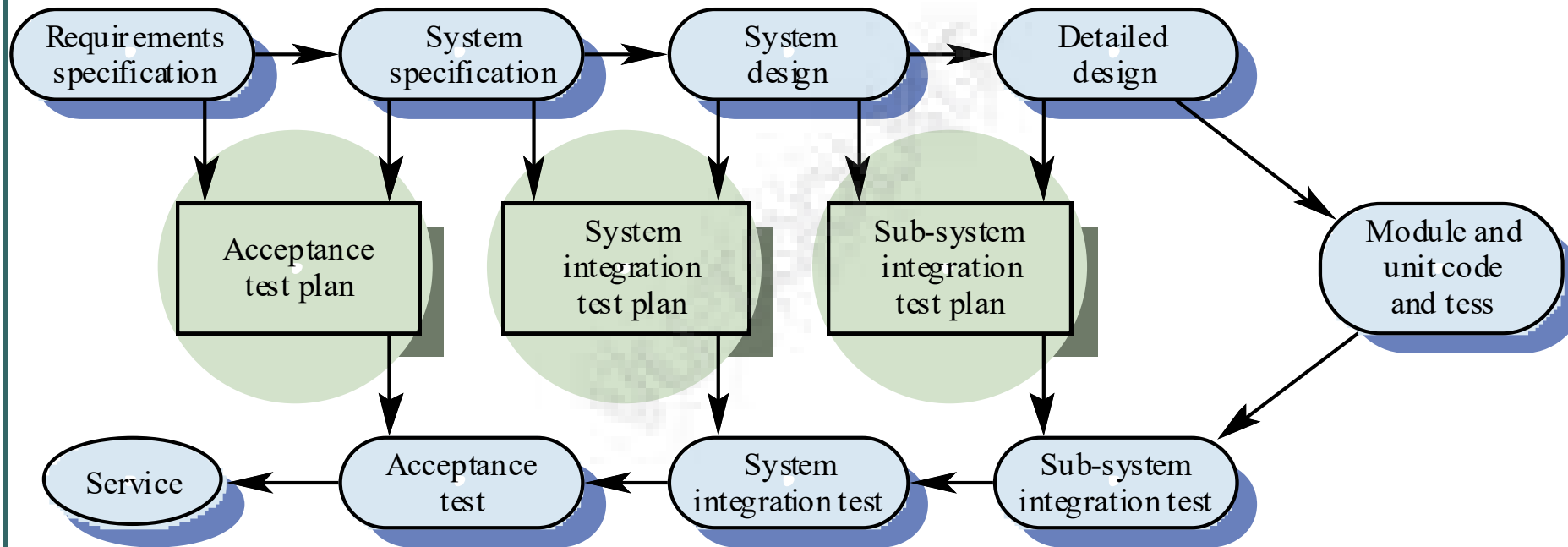
When to start testing...

- At the time of starting the projects
- How we start the projects ??
- Do we have any formal way ??





The V-model of development





Fact of testing

***Testing does not guarantee
the absence of defects***





What is test case

- A test case is a document that describes an input, action, or event and an expected response, to determine if a feature of an application is working correctly





Good test case design

- An good test case satisfies the following criteria:
 - Reasonable probability of catching an error
 - Does interesting things
 - Doesn't do unnecessary things
 - Neither too simple nor too complex
 - Not redundant with other tests
 - Makes failures obvious





Test case design technique

- Test case design techniques can be broadly split into two main categories
 - **Black box (functional)**
 - **White box (structural)**





Black Box tests

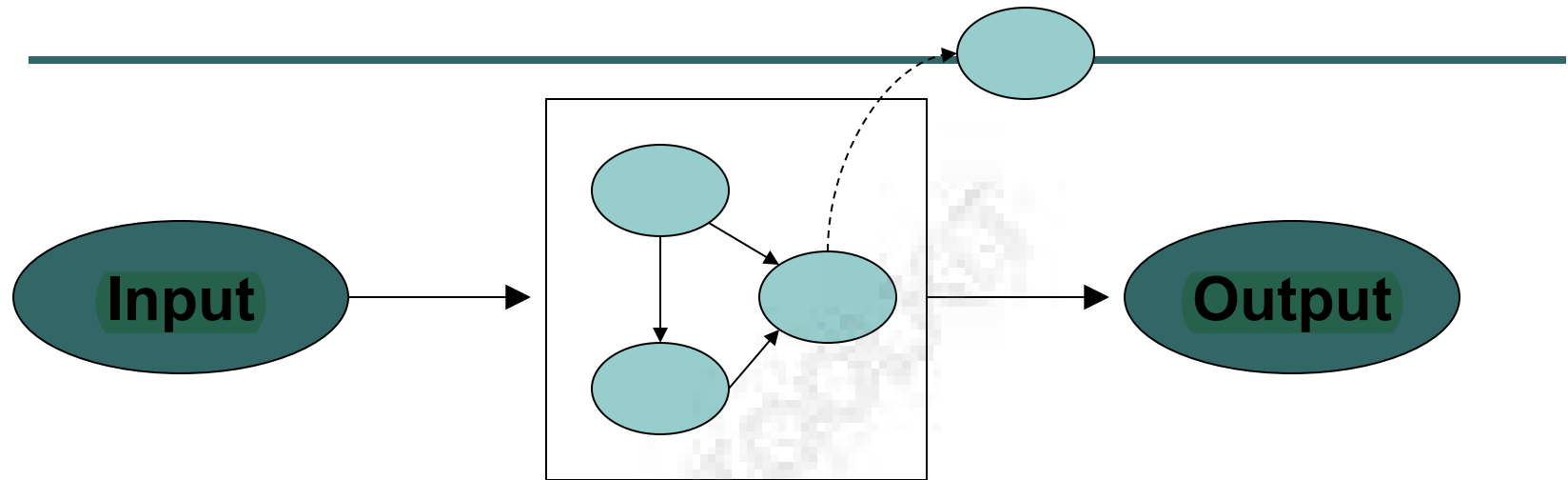


- Targeted at the apparent simplicity of the software
 - Makes assumptions about implementation
 - Good for testing component interactions
- Tests the interfaces and behavior





White Box tests



- Targeted at the underlying complexity of the software
 - Intimate knowledge of implementation
 - Good for testing individual functions
- Tests the implementation and design





Test case writing example

- Suppose we have two parameters we want to cover in a set of tests. Parameters are as follows..
 - Operating system
 - Win98
 - Win2k
 - Winxp
 - Printers
 - HP 4100
 - HP 4200

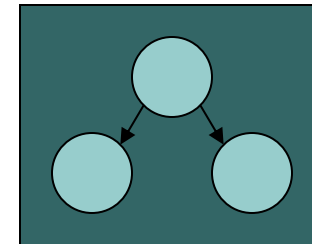
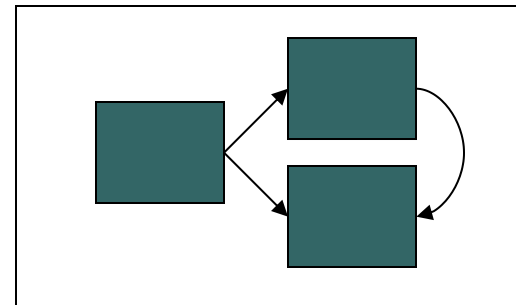
How We should write test case for this ??





Types of Tests

- Unit
 - Individual classes or types
- Component
 - Group of related classes or types
- Integration
 - Interaction between classes





What is a testing framework?

- A test framework provides reusable test functionality which:
 - Is easier to use (e.g. don't have to write the same code for each class)
 - Is standardized and reusable
 - Provides a base for regression tests





Why use a testing framework?

- Each class must be tested when it is developed
- Each class needs a regression test
- Regression tests need to have standard interfaces
- Thus, we can build the regression test when building the class and have a better, more stable product for less work





Testing tools

Tools are part of the quality equation, but not the entire equation





JUnit

- JUnit is a framework for writing unit tests
 - A **unit test** is a test of a *single* class
 - A **test case** is a single test of a single method
 - A **test suite** is a collection of test cases
- Unit testing is particularly important when software requirements change frequently
 - Code often has to be refactored to incorporate the changes
 - Unit testing helps ensure that the refactored code continues to work





JUnit..

- JUnit helps the programmer:
 - Define and execute tests and test suites
 - Formalize requirements and clarify architecture
 - Write and debug code
 - Integrate code and always be ready to release a working version





What JUnit does

- JUnit runs a suite of tests and reports results
- For *each* test in the test suite:
 - JUnit calls `setUp()`
 - This method should create any objects you may need for testing





What JUnit does...

- JUnit calls *one* test method
 - The test method may comprise multiple test cases; that is, it may make multiple calls to the method you are testing
 - In fact, since it's your code, the test method can do anything you want
 - The `setUp()` method ensures you *entered* the test method with a virgin set of objects; what you do with them is up to you
- JUnit calls `tearDown()`
 - This method should remove any objects you created





Creating a test class in JUnit

- Define a subclass of TestCase
- Override the `setUp()` method to initialize object(s) under test.
- Override the `tearDown()` method to release object(s) under test.
- Define one or more public `testXXX()` methods that exercise the object(s) under test and assert expected results.
- Define a static `suite()` factory method that creates a TestSuite containing all the `testXXX()` methods of the TestCase.
- Optionally define a `main()` method that runs the TestCase in batch mode.





Fixtures

- A fixture is just a some code you want run before every test
- You get a fixture by overriding the method
 - protected void **setUp()** { ... }
- The general rule for running a test is:
 - protected void **runTest()** {
 setUp(); <run the test> **tearDown();**
}
 - so we can override setUp and/or tearDown, and that code will be run prior to or after every test case





The structure of a test method

- A test method doesn't return a result
- If the tests run correctly, a test method does nothing
- If a test fails, it throws an `AssertionFailedError`
- The JUnit framework catches the error and deals with it; you don't have to do anything





Why JUnit

- **Allow you to write code faster while increasing quality**
- **Elegantly simple**
- **Check their own results and provide immediate feedback**
- **Tests is inexpensive**
- **Increase the stability of software**
- **Developer tests**
- **Written in Java**
- **Free**
- **Gives proper uniderstanding of unit testing**





Problems with unit testing

- JUnit is designed to call methods and compare the results they return against expected results
 - This ignores:
 - Programs that do work in response to GUI commands
 - Methods that are used primarily to produce output





Problems with unit testing...

- I think heavy use of JUnit encourages a “functional” style, where most methods are called to compute a value, rather than to have side effects
 - This can actually be a good thing
 - Methods that *just* return results, without side effects (such as printing), are simpler, more general, and easier to reuse



Code Reviews & Static Software Analysis

Types of reviews



Desk check

- Also called self check
- Informal review performed by the author of the artifact

Peer reviews

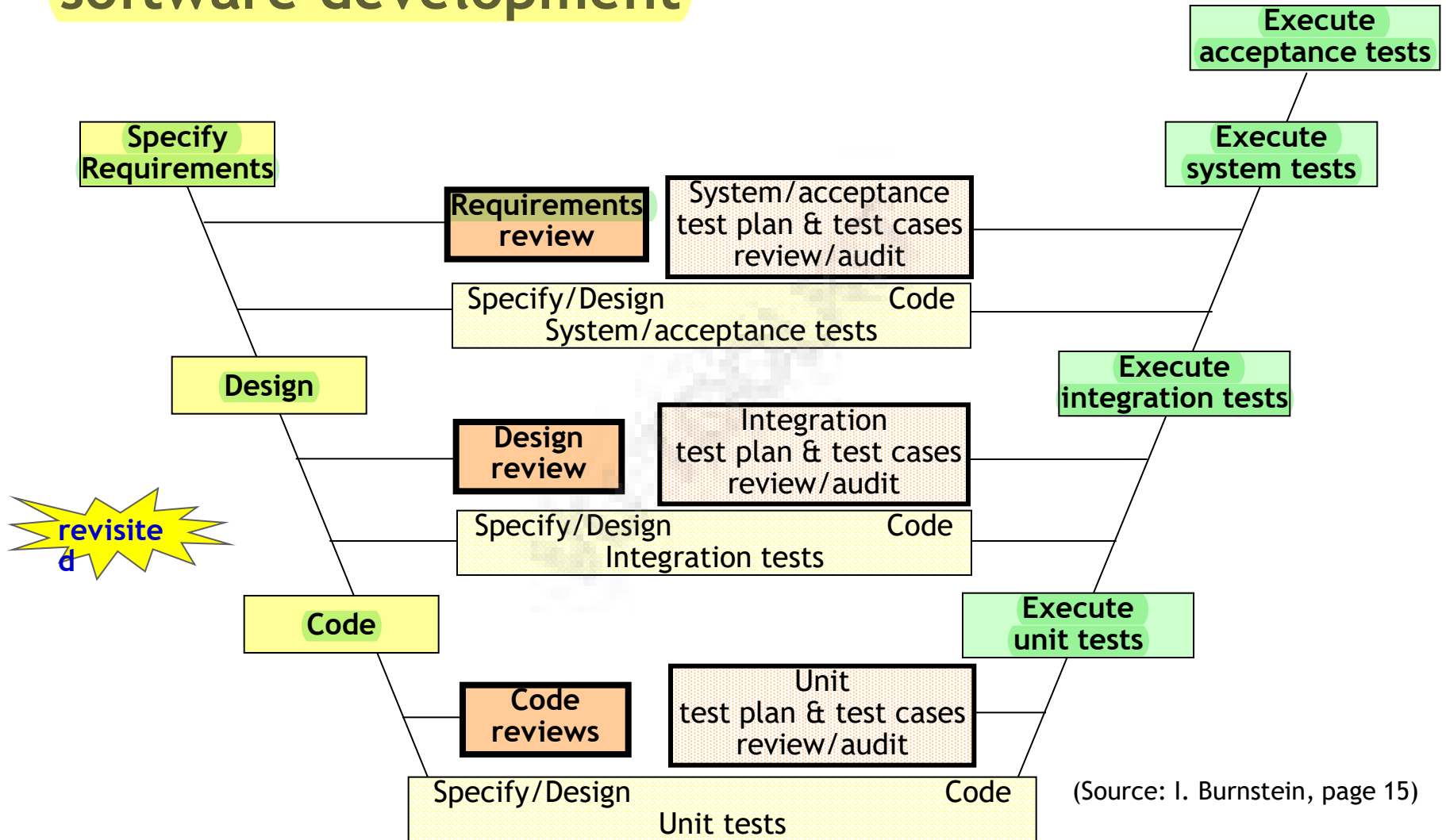
- ❑ “I show you mine and you show me yours”
- ❑ The author of the reviewed item does not participate in the review
- ❑ Effective technique that can be applied when there is a team (with two or more persons) for each role (analyst, designer, programmer, technical writer, etc.)
- ❑ The peer may be a senior colleague (senior/chief analyst, senior/chief architect, senior/chief programmer, senior/chief technical writer, etc.)

Audits

- ▣ An audit is an **independent evaluation of conformance** of software products and processes to applicable regulations, standards, plans, and procedures
- ▣ An audit is a formally organized activity, with participants having specific roles, such as lead auditor, other auditors, a recorder, an initiator, and a representative of the audited organization
- ▣ Audits may examine plans like recovery, SQA, design documentation, etc.
- ▣ Audits can occur on almost any product at any stage of the development or maintenance process

(source : SWEBOK)

Software reviews and the extended V-model of software development



(Source: I. Burnstein, page 15)

Typical tests and reviews

d

high-level design

(source: "Software Project Survival Guide", Steve McConnell)

Reviews and testing

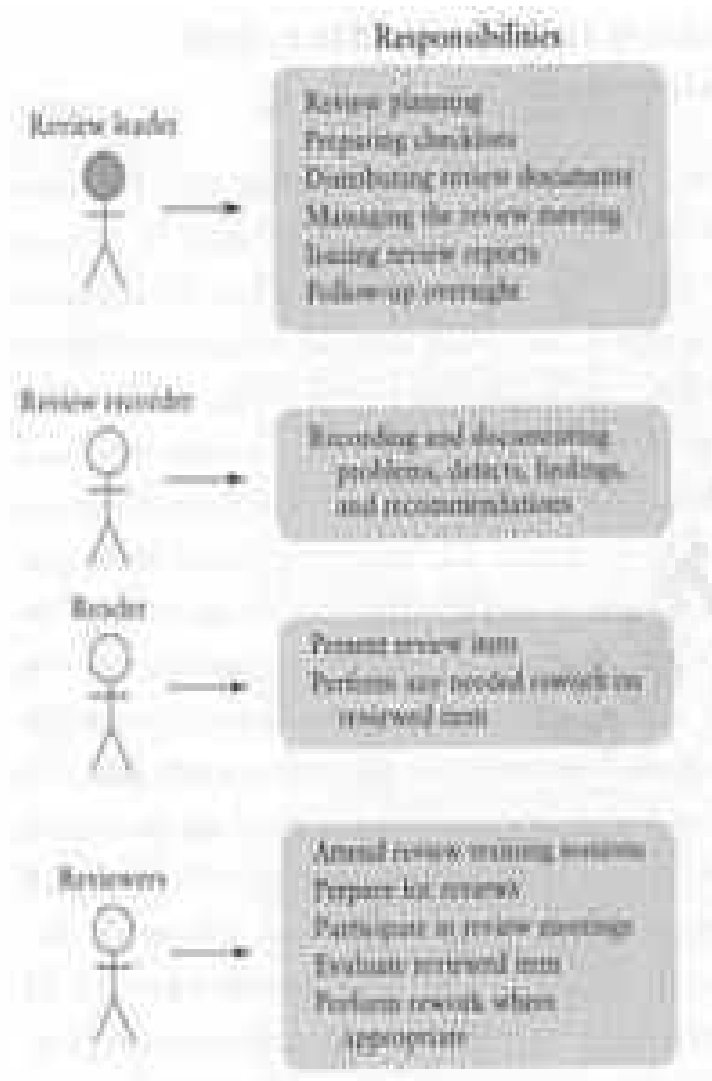
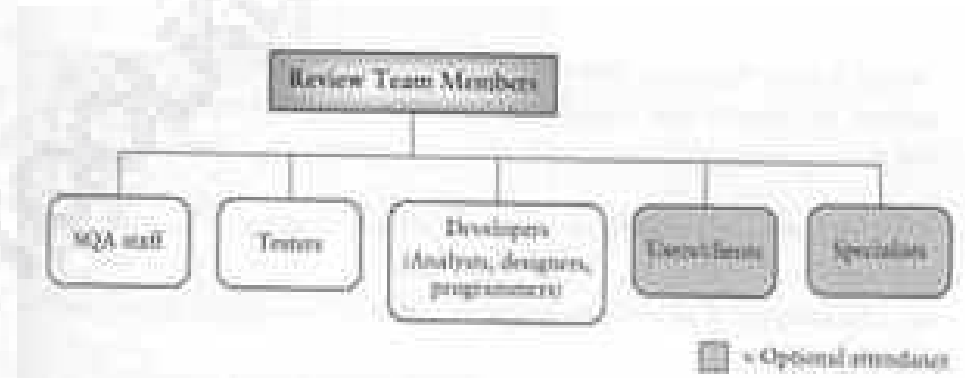
- ❑ A software system is more than the code; it is a set of related artifacts; these may contain defects or problem areas that should be reworked or removed; quality-related attributes of these artifacts should be evaluated
- ❑ Reviews allow us to detect and eliminate errors/defects *early* in the software life cycle (even before any code is available for testing), where they are less costly to repair
- ❑ Most problems have their origin in requirements and design; requirements and design artifacts can be reviewed but not executed and tested
 - Early prototyping is equally important to reveal problems in requirements and high-level architectural design
- ❑ A code review usually reveals directly the location of a bug, while testing requires a debugging step to locate the origin of a bug
- ❑ Adherence to coding standards cannot be checked by testing

Components of a review plan

- Review goals
- Items being reviewed
- Preconditions for the review
- Roles, team size, participants
- Training requirements
- Review steps and procedures
- Checklists and other related documents to be distributed to participants
- Time requirements
- Nature of the review log and summary report
- Rework and follow-up

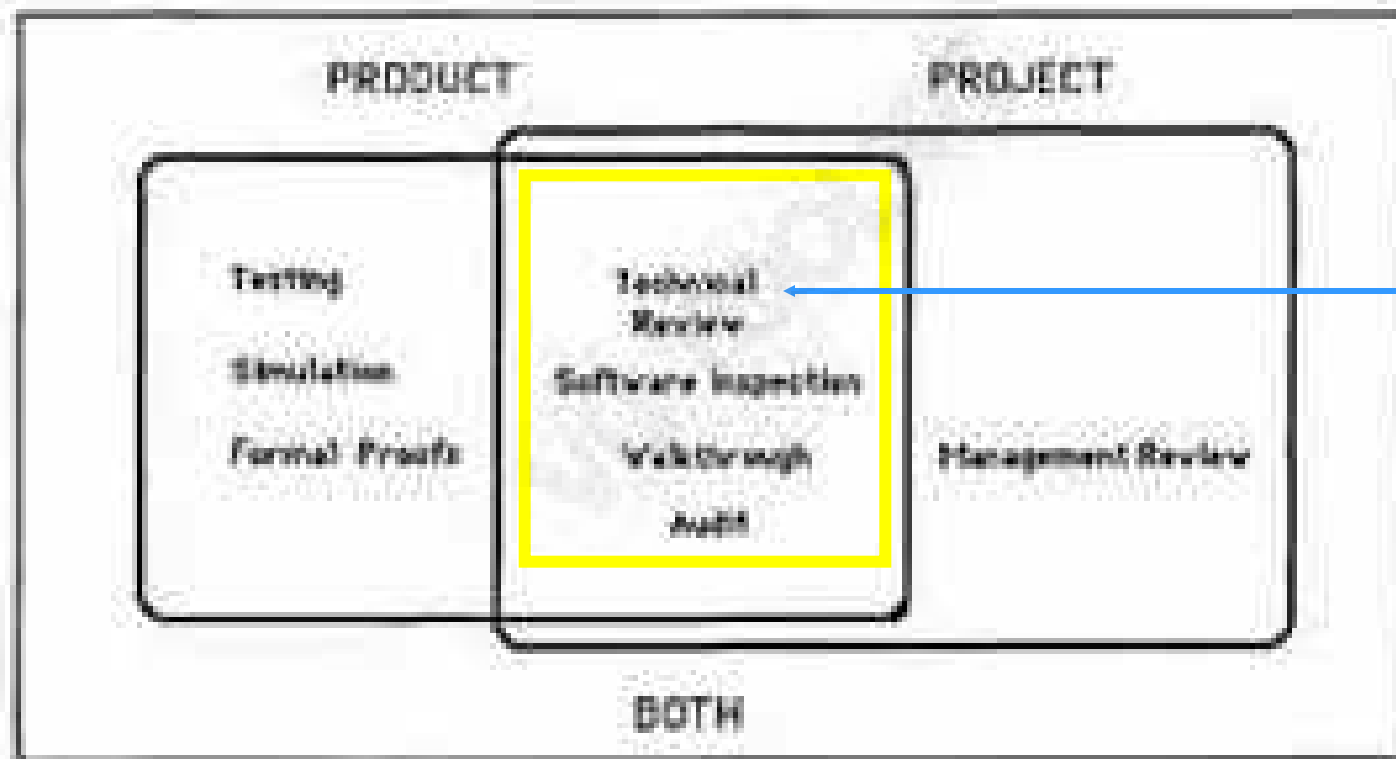
(Source: I. Bursntein)

Code Reviews & Static Software Analysis \$ Testing Techniques



IEEE Standard for Software Reviews and Audits (IEEE Std 1028-1988)

Fig 1
Relation of Some Quality Assurance Processes to Products and Projects



Specialized
meaning



Types of Reviews in IEEE Std 1028-1988

Category and Activities	Management Review	Technical Review	Subsystem Inspection	Walkthrough
Objective	Review progress. Recommend corrective action. Ensure proper allocation of resources.	Evaluate conformance to specifications and plans. Ensure design integrity.	Detect and identify defects. Verify resolution.	Detect defects. Examine alternatives. Forum for learning.
Delegated Controls				
Decision-making	Management team charts course of action. Decisions are made at the meeting or as a result of recommendations.	Review team petitions. Management or technical leadership makes recommendations.	Team chooses from predefined product dispositions. Defects must be resolved.	All decisions made by producer. Change is the prerogative of the producer.
Change verification	Change verification left to other project controls.	Leader verifies as part of review report.	Moderator verifies remark.	Change verification left to other project controls.
Group Dynamics				
Review-minded size	Two or more persons.	Three or more persons.	Three to six persons.	Two to seven persons.
Attendance	Management, technical leadership, and peer mix.	Technical leadership and peer mix.	College of peers most with documented attendance.	Technical leadership and peer mix.
Leadership	Usually the responsible manager.	Usually the lead engineer.	Trained moderator.	Usually producer.

A sample specification (or requirements) attributes checklist

Attribute	What to consider
Complete	Is anything missing or forgotten? Is it thorough? Does it include everything necessary to make it stand alone?
Accurate	Is the proposed solution correct? Does it properly define the goal? Are there any errors?
Precise, Unambiguous and Clear	Is the description exact and not vague? Is there a single interpretation? Is it easy to read and understandable?
Consistent	Is the description of the feature written so that it doesn't conflict with itself or other items in the specification?
Relevant	Is the statement necessary to specify the feature? Is there extra information that should be left out? Is the feature traceable to an original customer need?
Feasible	Can the feature be implemented with the available personnel, tools, and resources within the specified budget and schedule?
Code-free	Does the specification stick with defining the product and not the underlying software design, architecture, and code?
Testable	Can the feature be tested? Is enough information provided that a tester could create tests to verify its operation?

(Adapted from: Ron Patton, Software Testing)

A sample general code review checklist (1)

Design Issues

- Does each unit implement a single function?
- Are there instances where the unit should be partitioned?
- Is code consistent with detailed design?
- Does the code cover detailed design?

Data Items

- Is there an input validity check?
- Arrays-check array dimensions, boundaries, indices.
- Variables - are they all defined, initiated? have correct types and scopes been checked?
- Are all variables used?

Computations

- Are there computations using variables with inconsistent data types?
- Are there mixed-mode computations?
- Is the target value of an assignment smaller than the right-hand expression?
- Is over- or underflow a possibility (division by zero)?
- Are there invalid uses of integers or floating point arithmetic?
- Are there comparisons between floating point numbers?
- Are there assumptions about the evaluation order in Boolean expressions?
- Are the comparison operators correct?

A sample general code review checklist (2)

Control Flow Issues

- Will the program, module or, unit eventually terminate?
- Is there a possibility of an infinite loop, a loop with a premature exit, a loop that never executes?

Interface Issues

- Do the number and attributes of the parameters used by a caller match those of the called routine? Is the order of parameters also correct and consistent in caller and callee?
- Does a function or procedure alter a parameter that is only meant as an input parameter?
- If there are global variables, do they have corresponding definitions and attributes in all the modules that use them?

Input/output Issues

- Have all files been opened for use?
- Are all files properly closed at termination?
- If files are declared are their attributes correct?
- Are EOF or I/O errors conditions handed correctly?
- Is I/O buffer size and record size compatible?

A sample general code review checklist (3)

Portability Issues

- Is there an assumed character set, and integer or floating point representation?
- Are their service calls that may need to be modified?

Error Messages

- Have all warnings and informational messages been checked and used appropriately?

Comments/Code Documentation

- Has the code been properly documented? Are there global, procedure, and line comments where appropriate?
- Is the documentation clear, and correct, and does it support understanding?

Code Layout and White Space

- Has white space and indentation been used to support understanding of code logic and code intent?

Maintenance

- Does each module have a single exit point?
- Are the modules easy to change (low coupling and high cohesion)?

(Adapted from: Ilene Burnstein, page 331)

A sample code review checklist for C++ programs

(1)

Data Items

- Are all variables lowercase?
- Are all variables initialized?
- Are variable names consistent, and do they reflect usage?
- Are all declarations documented (except for those that are very simple to understand)?
- Is each name used for a single function (except for loop variable names)?
- Is the scope of the variable as intended?

Constants

- Are all constants in uppercase?
- Are all constants defined with a "#define"?
- Are all constants used in multiple files defined in an INCLUDE header file?

Pointers

- Are pointers declared properly as pointers?
- Are the pointers initialized properly?

A sample code review checklist for C programs (2)

 Control

- Are if/then, else, and switch statements used clearly and properly?

Strings

- Strings should have proper pointers.
- Strings should end with a NULL.

Brackets

- All curly brackets should have appropriate indentations and be matched

Logic Operators

- Do all initializations use an " = " and not an " = ="?
- Check to see that all logic operators are correct, for example, use of = / = =, and ||

Computations

- Are parentheses used in complex expressions and are they used properly for specifying precedences?
- Are shifts used properly?

(Adapted from: Ilene Burnstein, page. 331)

Types of (end-user) software documentation(1)

- ❑ **Packaging text and graphics.** Box, carton, wrapping, and so on. Might contain screen shots from the software, lists of features, system requirements, and copyright information.
- ❑ **Marketing material, ads, and other inserts.** These are all the pieces of paper you usually throw away, but they are important tools used to promote the sale of related software, add-on content, service contracts, and so on. The information for them must be correct for a customer to take them seriously.
- ❑ **Warranty/registration.** This is the card that the customer fills out and sends in to register the software. It can also be part of the software and display onscreen for the user to read, acknowledge, and even complete online.
- ❑ **EULA.** Pronounced "you-la," it stands for End User License Agreement. This is the legal document that the customer agrees to that says, among other things, that he won't copy the software nor sue the manufacturer if he's harmed by a bug. The EULA is sometimes printed on the envelope containing the media-the floppy or CD. It also may pop up onscreen during the software's installation.
- ❑ **Labels and stickers.** These may appear on the media, on the box, or on the printed material. There may also be serial number stickers and labels that seal the EULA envelope. See in a following slide an example of a disk label and all the information that needs to be checked.
- ❑ **Installation and setup instructions.** Sometimes this information is printed on the media, but it also can be included as a separate sheet of paper or, if it's complex software, as an entire manual.



Quality attributes (or dimensions) to check in technical information

Quality Dimension	Description
Task Orientation	
Task Orientation	Focus on the tasks that must be performed by the user to using the product.
Accuracy	Ensuring the information is correct, reliable and accurate.
Completeness	Ensuring the information contains all the necessary information.
User Centredness	
Clarity	Ensuring the information is clear and easy to understand.
Consistency	Ensuring the information is consistent and follows a logical structure.
Style	Ensuring the information is presented in a clear and concise manner.
Easy to Find	
Organization	Ensuring the information is organized in a logical and easy to follow manner.
Accessibility	Ensuring the information is accessible to all users, including those with disabilities.
Visual Effect/Support	Ensuring the information is presented in a visually appealing and easy to read manner.

Can be checked by asking probing questions, like:

- Is the information appropriate for the intended audience?
- Is information presented from a user's point of view?
- Is there a focus on real tasks?
- Is the reason for the information evident?
- Do titles and headings reveal real tasks?

Build your own check list!
Adapt to your needs!

Source: **Developing Quality Technical Information (DQTI)**, Hargis, IBM, 1997

not only for software, not only for end-user documentation (also for developers and maintainers)

Contents of a formal review report (1)

- Checklist will all items covered (with a check mark) and comments relating to each item
- List of defects found, with
 - description
 - type
 - frequency
 - defect class, e.g.
 - missing
 - incorrect
 - superfluous
 - location
 - cross-reference to the place or places in the reviewed document where the defect occurs
 - severity, e.g.
 - major
 - minor

Contents of a formal review report (2)

- Summary report, with

- list of attendees
- review metrics, such as
 - number of participants
 - duration of the meeting
 - size of the item being reviewed (usually LOC or number of pages)
 - number of defects found
 - total preparation time for the review team
 - number of defects found per hour of review time
 - number of defects found per page or LOC
 - LOC or pages reviewed per hour
 - ...
- status of the reviewed item (requirements document, etc.)
 - **accept** - the item is accepted in its present form or with minor rework required that does not need further verification
 - **conditional accept** - the item needs rework and will be accepted after the moderator has checked and verified the rework
 - **reinspect** - considerable rework must be done to the item. The inspection needs to be repeated when the rework is done.
- estimate of rework effort and the estimated date for completion of the rework
- signatures and date

Thank You