IT3205: Fundamentals of Software Engineering (Compulsory)

BIT – 2nd Year

Semester 3

IT3205: Fundamentals of Software Engineering

Software Design

Duration: 8 hours

Learning Objectives

- Describe the important software design issues and concepts.
- Compare different approaches to software design.
- Identify suitable design approaches for a problem.

Detailed Syllabus

4.1 Design concepts

- 4.1.1 Abstraction
- 4.1.2 Architecture
- 4.1.3 Patterns
- 4.1.4 Modularity
  - 4.1.4.1 Cohesion
  - 4.1.4.2 Coupling
- 4.1.5 Information hiding
- 4.1.6 Functional independence
- 4.1.7 Refinement

Detailed Syllabus

4.2 Architectural design

- 4.2.1 Repository model
- 4.2.2 Client-server model
- 4.2.3 Layered model
- 4.2.4 Modular decomposition

4.3 Procedural design using structured methods

Detailed Syllabus

4.4 User Interface design

4.4.1. Human-computer interaction

4.4.2. Information presentation

4.4.3. Interface evaluation

4.5 Design notations

What is Software Design ?

- Design is the process of translating the requirements in to a meaningful engineering representation that can be implemented.
- In the context of software engineering, design focuses on transforming the requirements into an *implementable version* of the software system.
- Design stage has the greatest influence on software quality.

Software Design – Why it is important?

- A good design
  - is the key for a successful software system.
  - allows easy maintenance of a system.
  - allows to achieve non-functional requirements such as reliability, performance, reusability and portability.
  - facilitates the development and management processes of a software project.

Design activities

- Identification of the software architecture
- Architectural design
  - Identification of the sub systems and components
- Data design
  - Organizing the data so as to facilitate effective utilization
- Procedural design
  - How inputs are transformed into outputs
- Interface design
  - How to make the system user friendly
- Algorithm design
- Design specification

4.1 DESIGN CONCEPTS

Design Concepts

- **Abstraction**
  - Permits one to concentrate on a problem at some level of generalization without regard to irrelevant low level details
- **Stepwise refinement and partitioning**
  - Increasing level of detail by allocating functionality to modules
- **Modularity**
  - Self contained and loosely coupled software components
- **Information Hiding & Encapsulation**
  - Modules should be specified and designed so that code / data contained in a module is directly inaccessible to other modules. Protecting information from direct access by other modules and providing access to this information through well defined interfaces is called Encapsulation.
- **Polymorphism**
  - Modules should be flexible and promote reusability

Abstraction

- This is an intellectual tool (a psychological notion) which permits one to concentrate on a problem at some level of generalization without regard to irrelevant low level details
- Abstraction allows us to proceed with the development work without been held up in low-level implementation details (yet to be discovered)
- Two forms of abstraction
  - Procedural abstraction
  - Data abstraction

Abstraction

Example:

Develop software that will perform 2-D drafting (CAD)

Abstraction 1

Software will include a computer graphics interface which will enable the draftsperson to see a drawing and to communicate with it via a mouse. All line and curve drawing, geometric computations will be performed by the CAD software. Drawing will be stored in a drawings file.

Abstraction

Abstraction 2

CAD software tasks:

user interaction task;
2-D drawing task;
graphics display task;
drawing file management task;

End.

Procedural Abstraction

Data Abstraction = defining a data object at different levels.

Software Architecture

- Overall structure of the software components and the ways in which that structure provides conceptual integrity for a system.

Design Patterns

- When reusing software components, the developer is constrained by the design decisions that have been made by the implementers of these components.
- If the design decisions of the reusable components conflicts whit the current requirements the reusability will be impossible or the developed system will become inefficient.
- One way to solve the problem is to reuse more abstract designs that do not include implementation details

Modularity

- Software is divided into separately named, addressable components called modules.
- The complexity of a program depends on modularity.

Let C(x) = a measure of complexity and P1 and P2 be problems,

E(x) = a measure of effort to solve

If C(P1) > C(P2) then

E(P1) > E(P2)

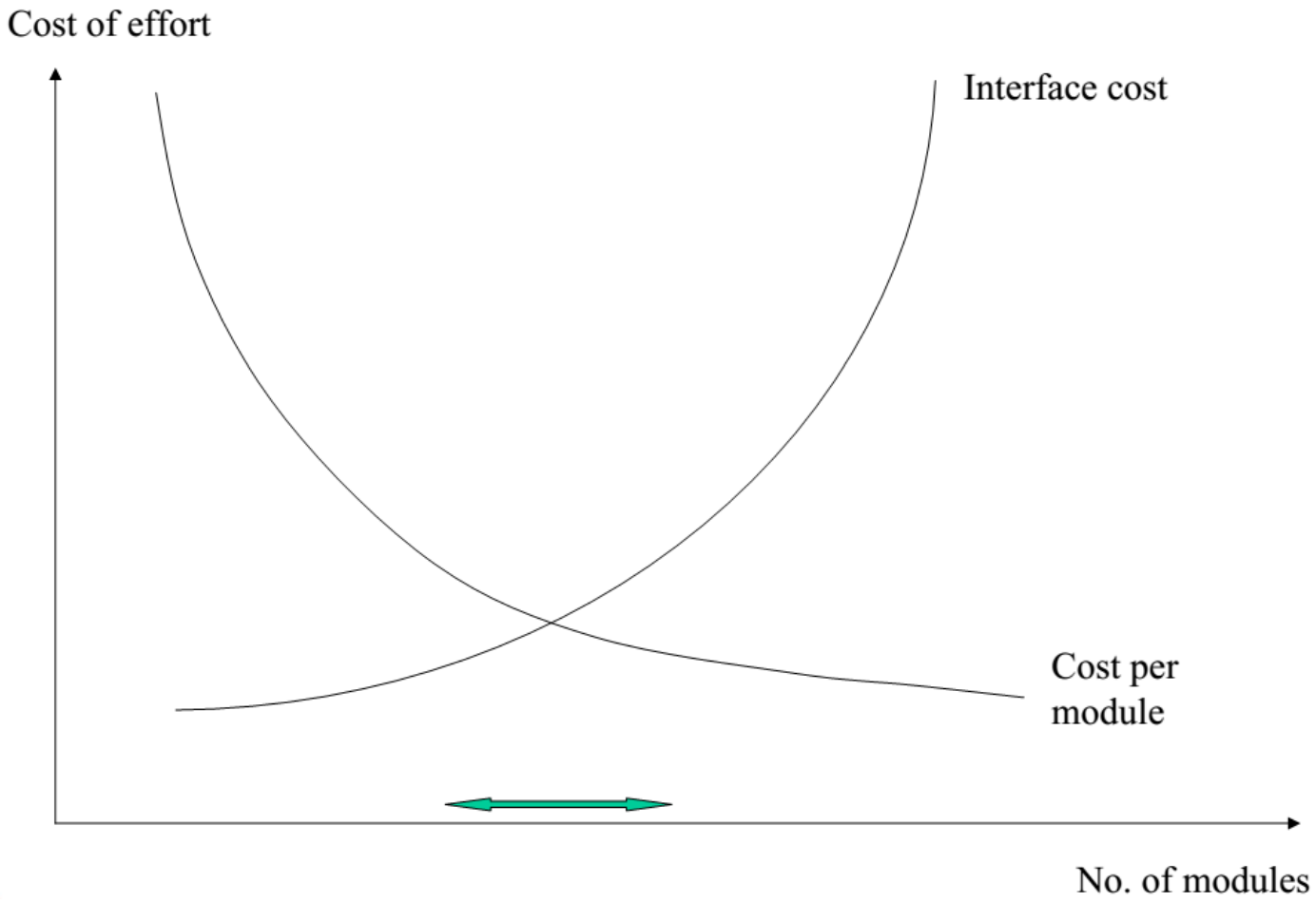
Also, C(P1+P2) > C(P1) + C(P2)

Therefore, E(P1+P2) > E(P1) + E(P2)

Modularity

- Modularity facilitates
  - the development process
  - the maintenance process
  - the project management process
  - reusability

How many modules?



Module Coupling

- A measure of the strength of the interconnections between system components.
- Loose coupling means component changes are likely to affect other components.
- Shared variables or control information exchange lead to tight coupling.
- Loose coupling can be achieved by component communication via parameters or message passing.

Levels of Coupling

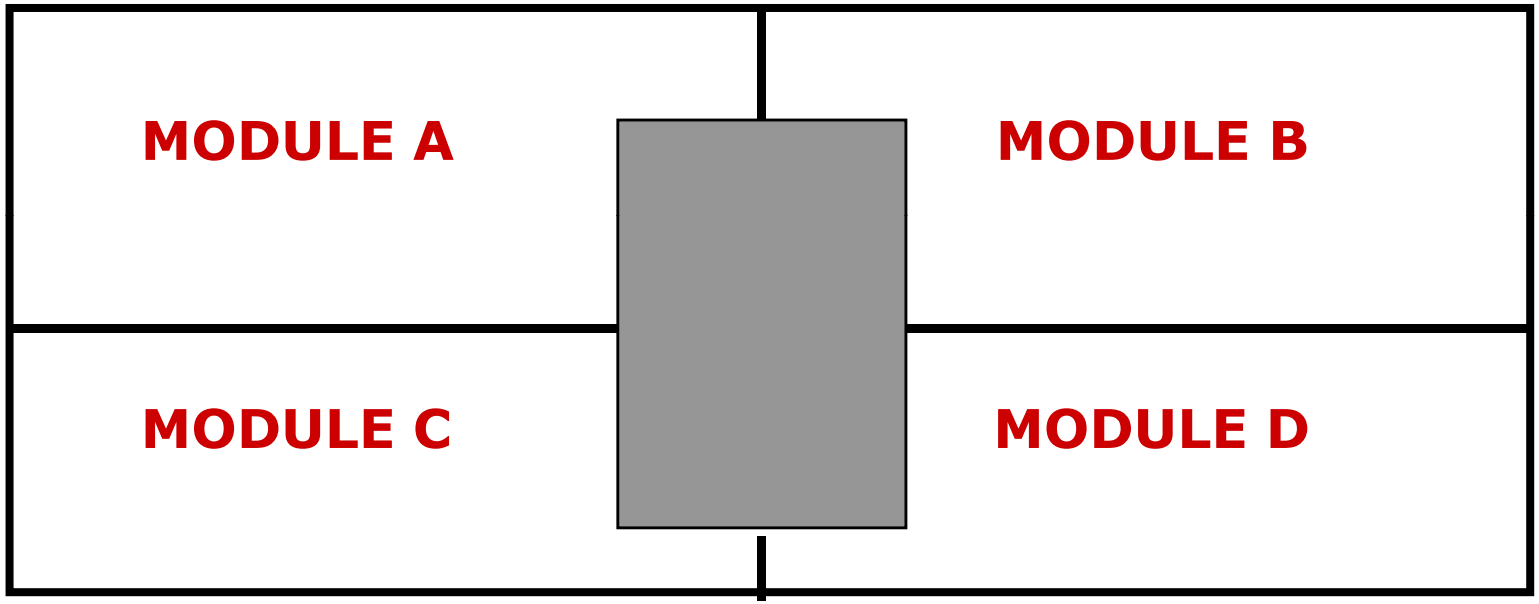
- **Data Coupling**
  - Data is passed from one module to another using arguments
- **Stamp Coupling**
  - More data than necessary is passed via arguments.

    Eg. Pass the whole record instead of just the field being changed.
- **Control Coupling**
  - A flag is passed from one module to another affecting the functionality of the second module
- **External Coupling**
  - Coupling with the environment

    (eg. Data files, other programs etc.).

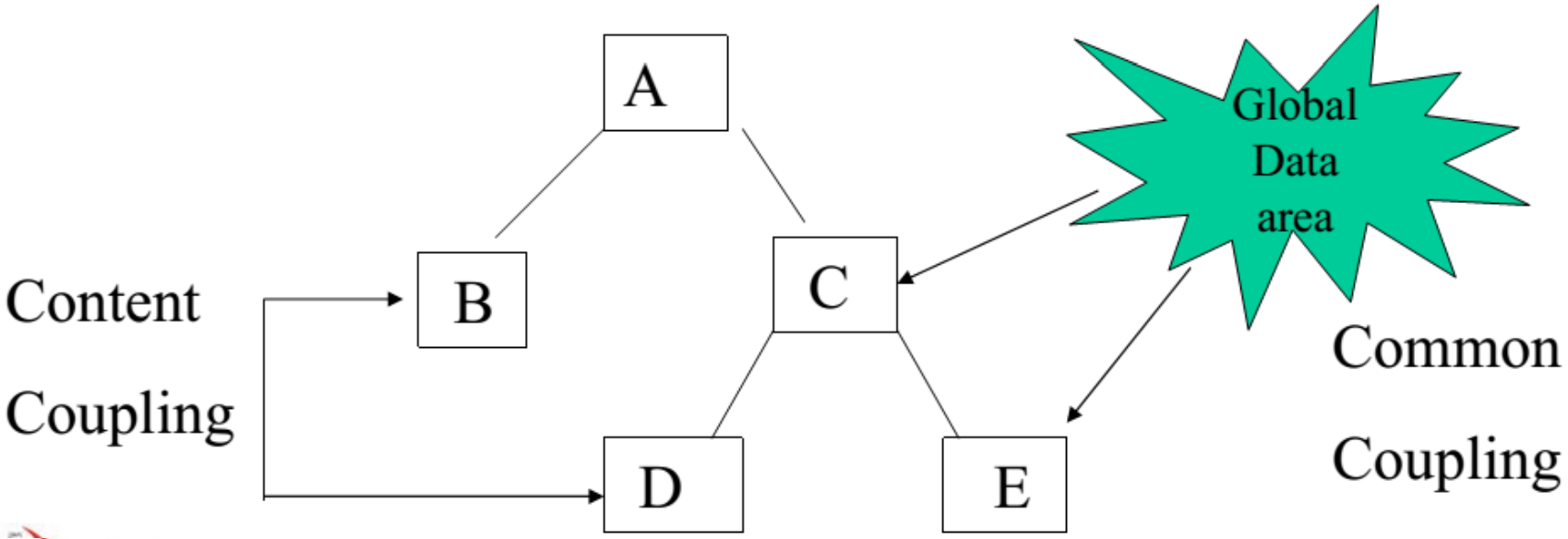
Levels of Coupling

- **Common Coupling**
  - Occurs when modules access the same global data



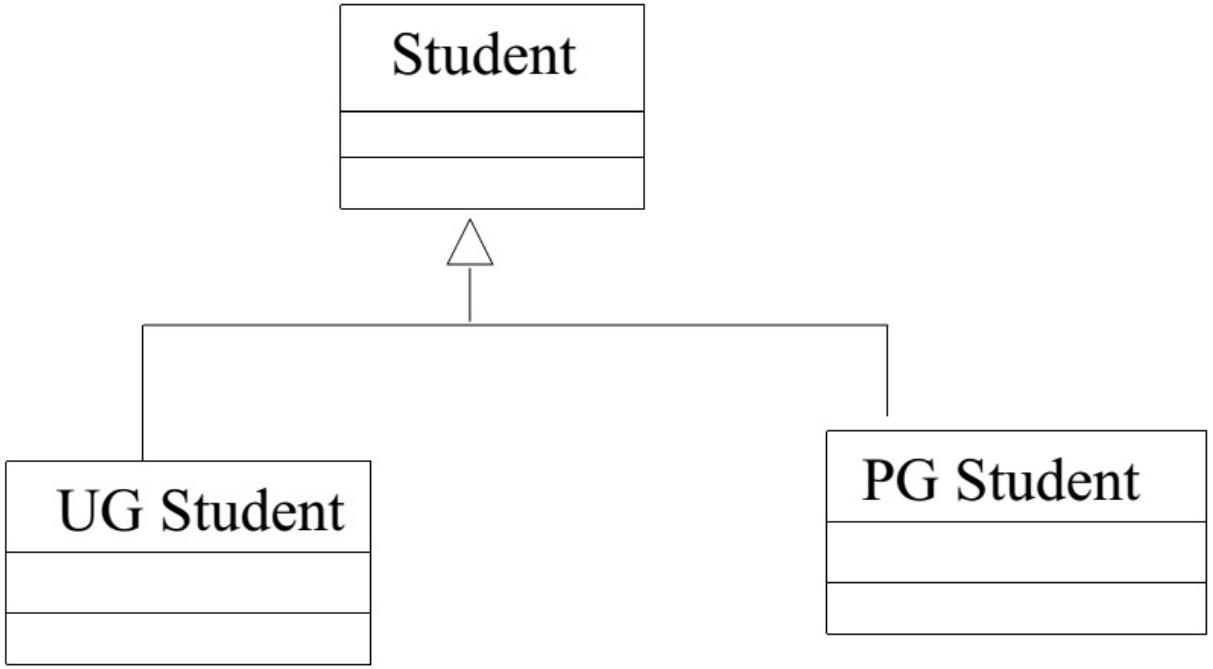
Levels of Coupling

- **Content Coupling**
  - One module directly affects the working of another. Calling module can modify the called module or refer to an internally defined data element.



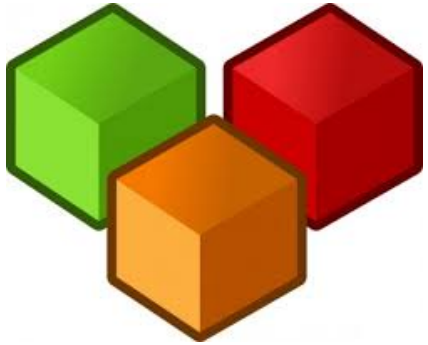
Levels of Coupling

- **Object Coupling**
  - Object oriented systems are loosely coupled. No shared state and objects communicate using message passing. Object coupling occurs when a class inherits attributes and methods of another class.



Coupling

- Coupling should be minimized.
- Loosely coupled modules facilitate:
  - Maintenance
  - Development
  - Reusability



Module Cohesion

- Interaction within a module. A measure of how well a component fits together.
- High cohesion – A component should implement a single logical entity or function
- Cohesion is a desirable design component attribute as when a change has to be made, it is localized in a single component.

Levels of Cohesion

- **Object Cohesion**
  - Occurs when a single entity is represented by the object and all operations on the object, and no others are included within it. This is the strongest type of cohesion and should be aimed by the designer.
- **Functional Cohesion**
  - Occurs when all the elements of the module combine to complete one specific function. This also strong cohesion and should be recommended.
- **Sequential Cohesion**
  - Occurs when the activities (more than one purpose to the function) combine such that the output of one activity is the input to the next. Not as good as functional cohesion but still acceptable.

Levels of Cohesion

- **Communicational Cohesion**
  - Occurs when a module performs a number of activities on the same input or output data.

    For example customer maintenance functions in a business information system such as add customer, delete customer, update customer details and print customer details exhibits communicational cohesion because they all operate on a customer file.

Levels of Cohesion

- **Procedural Cohesion**
  - Occurs when a modules' internal activities bear little relationship to one another but control flows one to another in sequence.

```
class MakeCake {
    void addIngredients() { ... }
    void mix() { ... }
    void bake() { ... } }
```

Levels of Cohesion

- **Temporal Cohesion**
  - Occurs when functionality is grouped simply because it occurs at the same time. For example house keeping tasks at the start and end of an application.

```
class InitFuns {
   void initDisk() { ... }
   void initPrinter() { ... }
   void initMonitor() { ... } }
```

Levels of Cohesion

- **Logical Cohesion**
  - Occurs when functionality is grouped by type. For example all creates together, all updates together etc. This should be avoided at all cost.

```
class AreaFuns {
    double circleArea() { ... }
    double rectangleArea() { ... }
    double triangleArea() { ... }}
```

Levels of Cohesion

- **Coincidental Cohesion**
  - Occurs when functionality is grouped randomly. Not even to be considered as an option in design.

```
class MyFuns {
   void initPrinter() { ... }
   double calcInterest() { ... }
   Date getDate() { ... }}
```

Information Hiding

- The principle of Information Hiding suggests that modules be characterized by design decisions that (each) hides from all others. In other words modules should be specified and designed so that information (procedure and data) contained within a module is directly inaccessible to other modules.
- However the modules should communicates using well defined interfaces. Protecting information from direct access by other modules and providing access to this information through well defined interfaces is called **Encapsulation**.
- Because most data and procedure are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

Encapsulation

- Encapsulation is a technique for minimizing interdependencies among separately written modules by defining strict external interfaces.
- The external interface acts as a contract between a module and its clients.
- If clients only depend on the interface, modules can be re-implemented without affecting the client.
- Thus the effects of changes can be confined.

Functional Independence

- Achieved by developing modules with single-minded purpose and an aversion to excessive interaction with other models

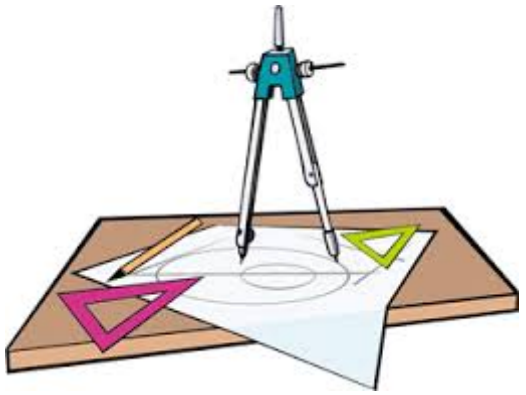
Refinement

- Process of elaboration where the designer provides successively more detail for each design component

Design with reuse

- Another important design consideration is reusability of software components. Designing reusable software components is extremely valuable.
- Some benefits of software reuse are;
  - Increased reliability
  - Reduced process risks
  - Effective use of specialists
  - Standards compliance
  - Accelerated Development

4.2 ARCHITECTURAL DESIGN



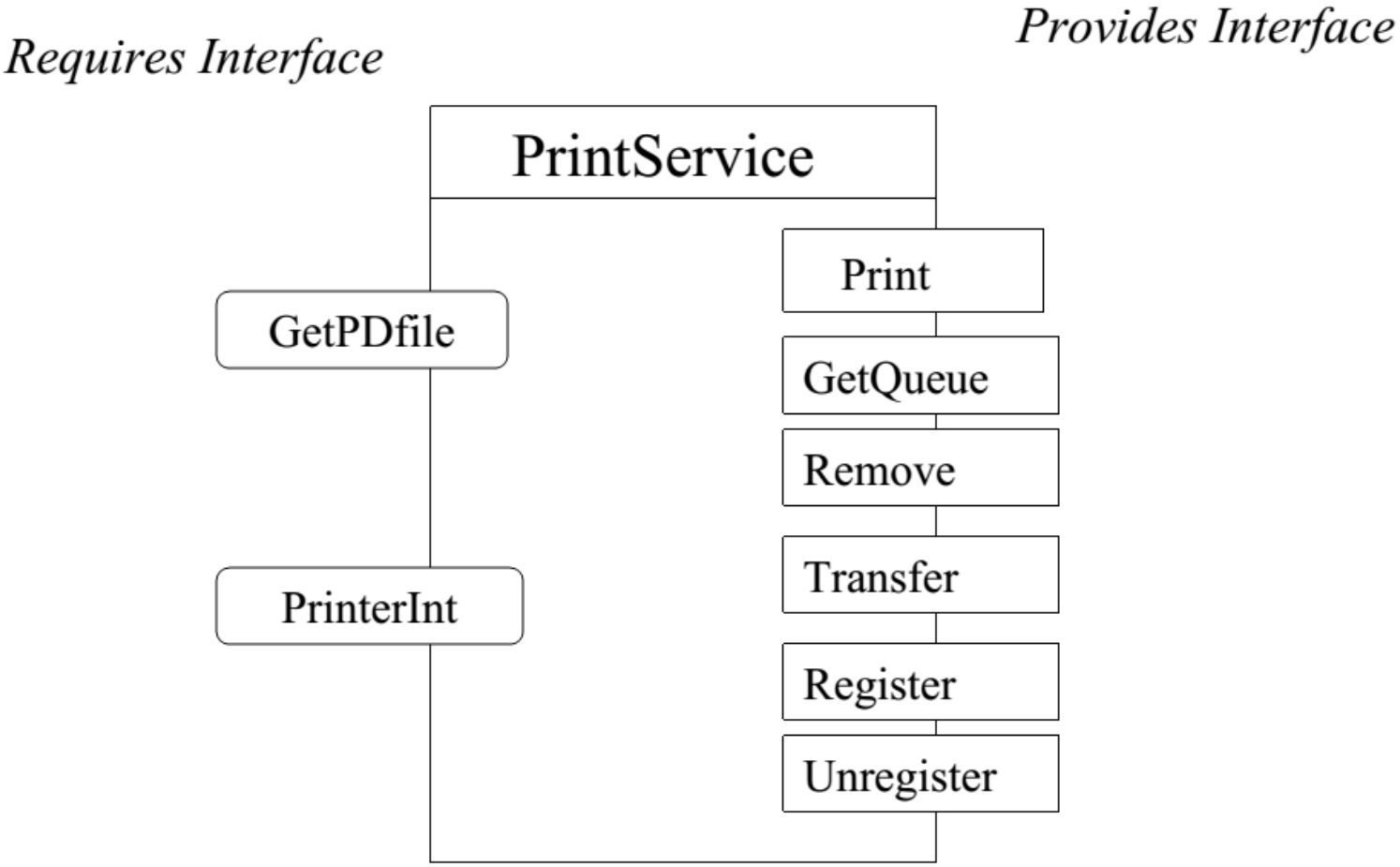
Software Architectural Design

- The architectural design process is concerned with establishing a basic structured framework for a system. It involves identifying the major components of the system and the communications between these components.
- Large systems are always decomposed into subsystems that provide some related set of services. The initial design process of identifying these sub-systems and establishing a framework for subsystem control and communication is called *architectural design* and the output of this design process is a description of the *software architecture*.

Sub-Systems and Components

- **Sub- Systems-** A Sub-system is a system in its own right whose operations are not depend on the services proved by the other sub-systems. Sub systems are composed of modules (components) and have defined interfaces which are used for communication with other sub-systems.
- **Components–** A component (module) is normally a system component that provides one or more services to other modules. It makes use of services provided by other modules. It is not normally considered as an independent system. Modules are usually composed from a number of other, simpler system components.

Software components - An Example



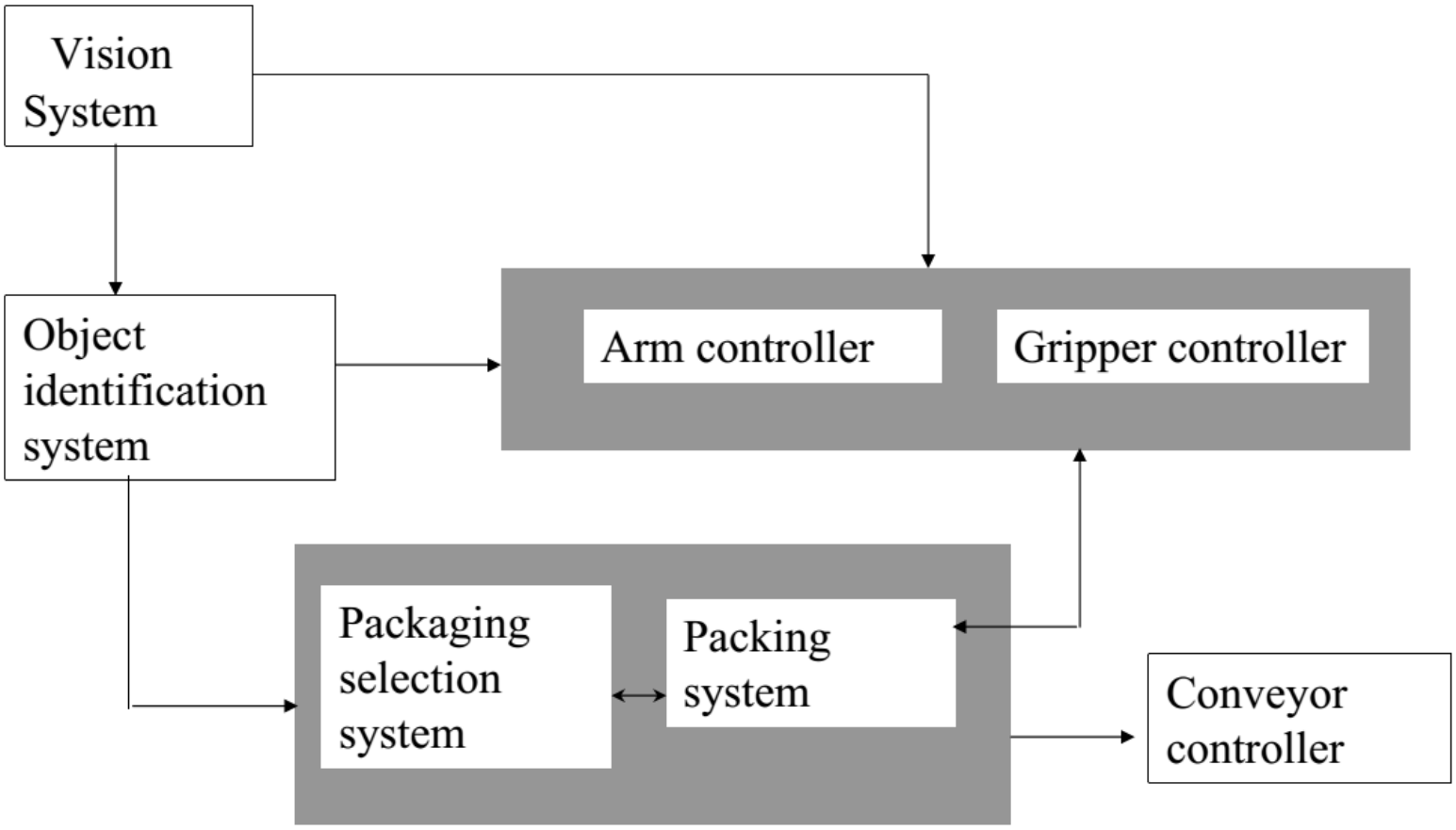
Software components - An Example

- **GetPDfile** –A service to retrieve the printer description file for a printer type
- **PrinterInt** –A service that transfers commands to a specified printer.
- **Print** –A service to print a document
- **GetQueue** –Discover the state of a print queue
- **Remove** –Remove a job from the queue
- **Transfer** –Transfer a job to another queue
- **Register** -Register a printer with the printing service component
- **Unregister** –unregister a printer

Architectural Design Process

- **System structuring**
  - The system is structured into a number of principal sub-systems where a sub-system is an independent software unit. Communications between sub-systems are identified.
- **Control modelling**
  - A general model of the control relationships between the parts of the systems is established.
- **Modular decomposition**
  - Each identified sub-system is decomposed into modules. The architect must decide on the types of module and their interconnections.

Architectural Design – An example



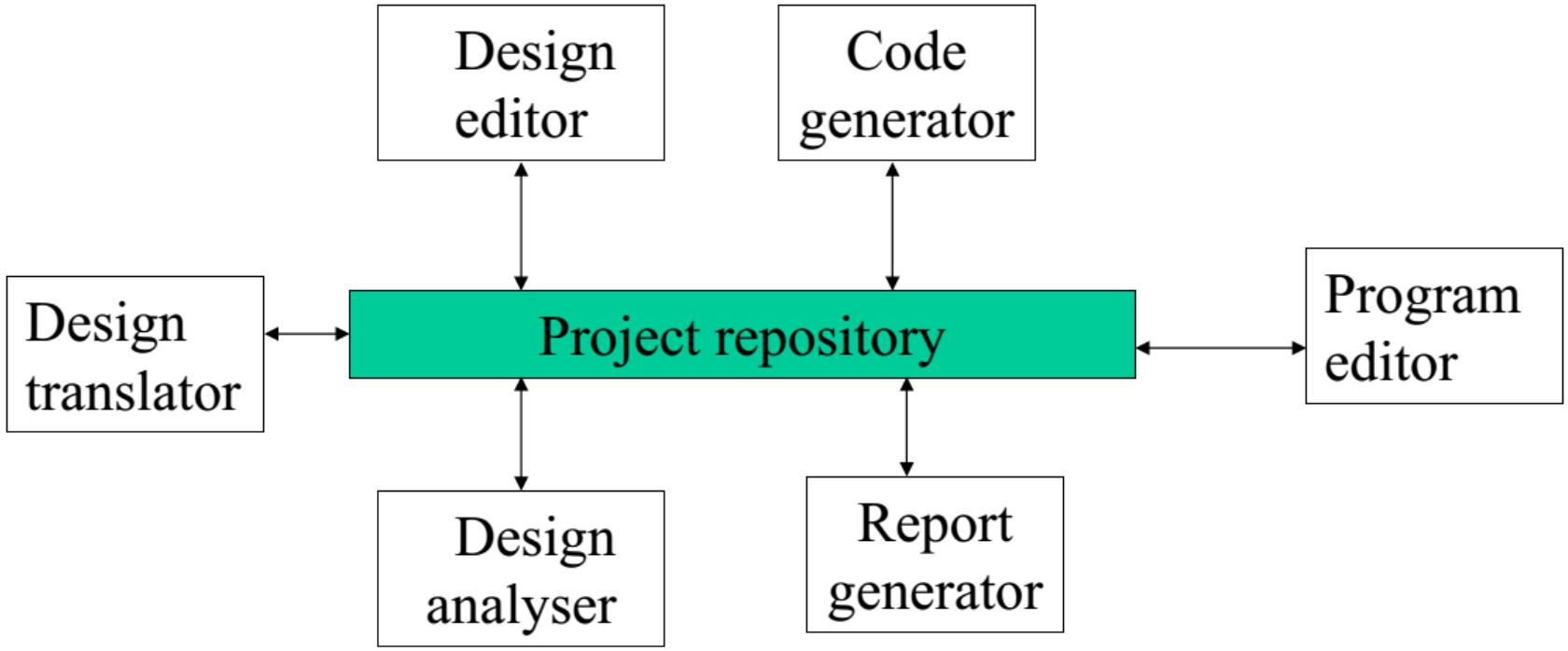
Repository Model

- Sub-systems making up a system must exchange information so that they can work together effectively. One approach is to keep all shared data in a central database that can be accessed by all sub-systems. A system model based on a shared database is called *repository model*.

- This model is suited to applications where data is generated by one sub-system and used by another. Examples of this type of systems include command and control systems, management information systems CAD systems and CASE tools.

Repository Model - An example

The architecture of an integrated CASE tool.

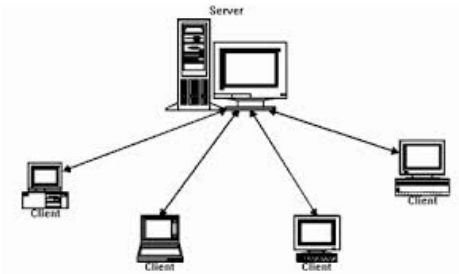


Repository Model - Advantages

- It is efficient way to share large amount of data. There os no need to transmit data explicitly from one sub-system to another.
- Activities such as backup recovery, access control and recovery from error are centralized. They are the responsibility of the repository manager. Tools can focus on their principal function rather than be concerned with these issue.
- The model of sharing is visible through the repository schema. It is straight forward to integrate new tools given that they are compatible with the agreed data model.

Repository Model - Disadvantages

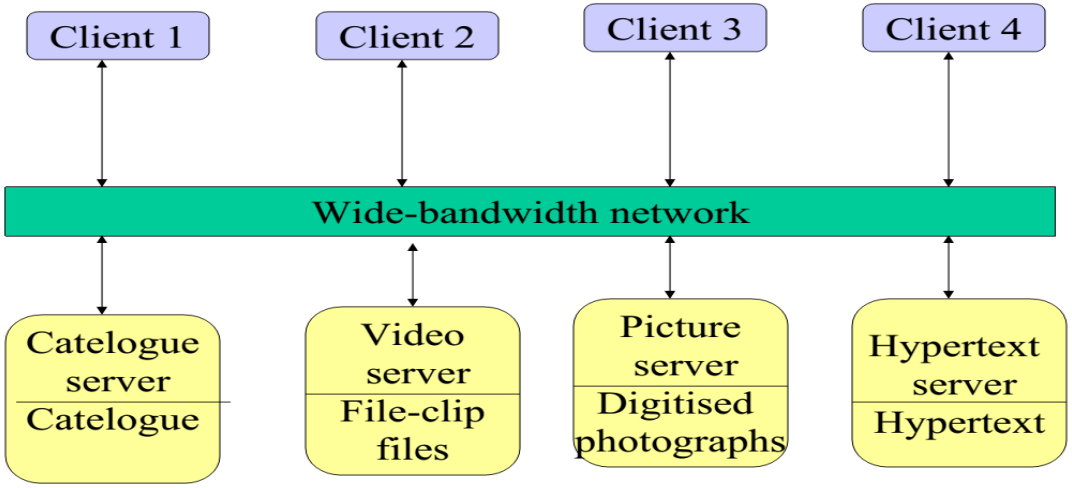
- Sub-systems must agree on a repository data model. Inevitably, this is a compromise between the specific needs of each tool. Performance may be adversely affected by this compromise. It may be difficult or impossible to integrate new sub-systems if their data models do not fit the agreed schema.
- Evolution may be difficult as a large volume of information is generated according to an agreed data model. Translating this to a new model will certainly be expensive.
- Different sub-systems may have different requirements for security, recovery and backup policies. The repository model forces the same policy on all sub-systems.



Client-Server Model

- The client-server architectural model is a distributed system model which shows how data and processing are distributed across a range of processors.
- The major components of the model are:
  1. A set of stand-alone servers which offer services to other subsystems. Examples of servers are print servers, web servers and data base servers.
  2. A set of clients that call on the services offered by the servers. Theses are normally sub-systems in their own right. There may be several instances of a client program executing concurrently.
  3. A network which allows the clients to access these services.

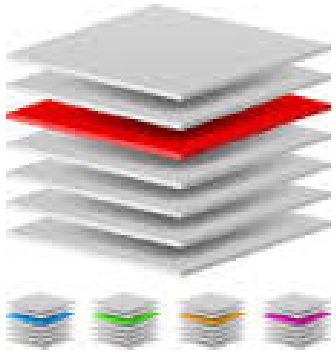
Client-Server Model – An Example



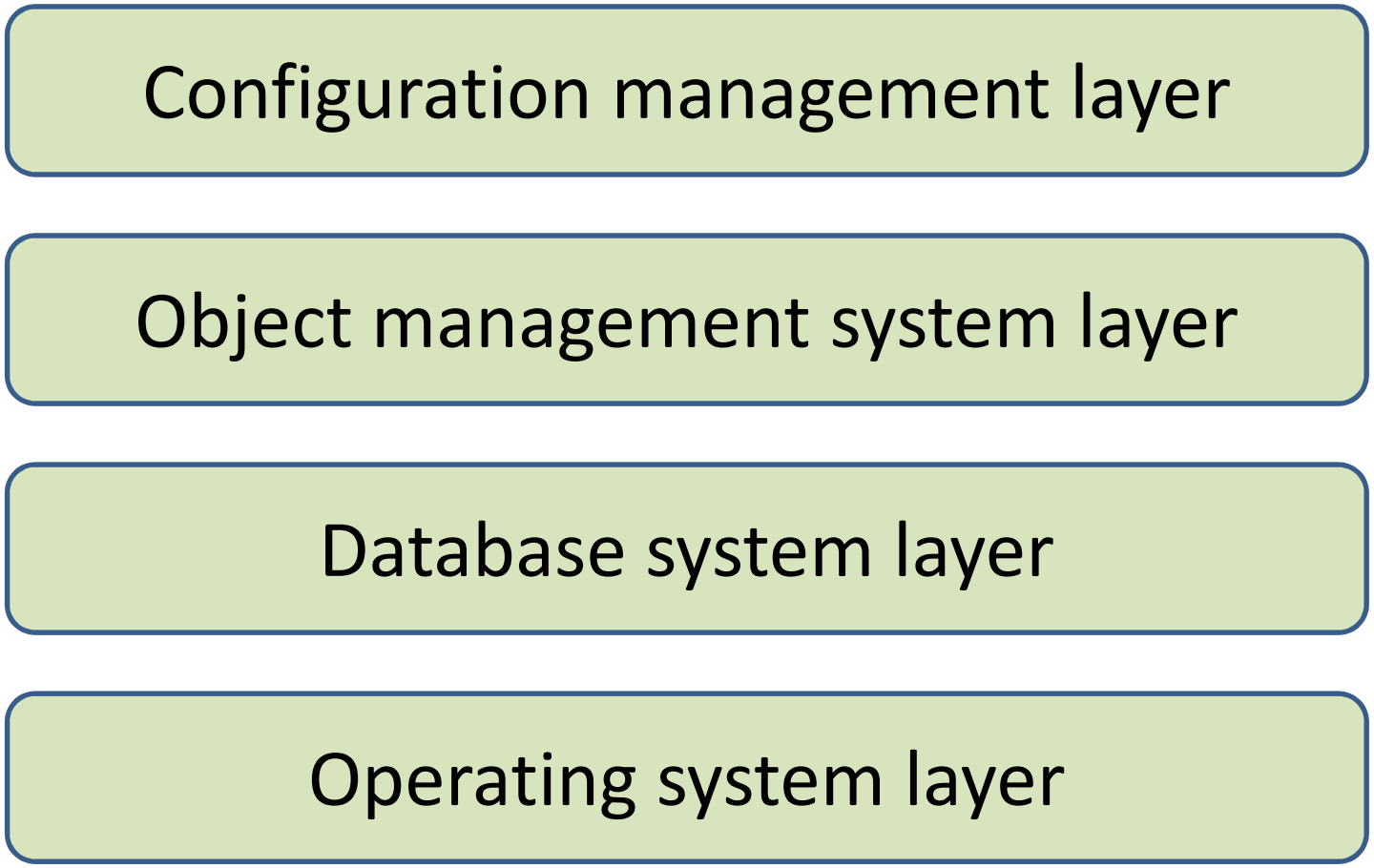
The above system is multi-user hypertext system to provide a film and photograph library. In this system, there are several severs which manage and display the different type of media. Video frames need to be transmitted quickly and in synchrony but at relatively low resolution. They may be compressed in a store. Still pictures, however, must be sent at a high resolution. The catalogue must be able to deal with a variety of queries and provide links into the hypertext information systems. The client program is simply an integrated user interface of these services.

Layered Model

- The layered model organizes the system in to layers by modeling the interfacing of sub systems.
- Each layer provides a set of services.
- This model is also called abstract machine model.
- Each layer defines an abstract machine language is used to implement the next level of abstract machine.



Layered model of a version management system



Layered Model

- An example for layered model is OSI reference model for network protocols.
- This model support the incremental development of systems because some of the services provided by the layer is made available to users while the layer is being developing.

Modular Decomposition

- After a structural architecture has been designed, the next stage of architectural design process is the decomposition of sub-system in to modules.
- There is not a rigid distinction between system decomposition and modular decomposition.

Modular Decomposition

- We consider two modules which may be used when decomposing a sub-system into modules.

1. **An object oriented model**
   - The system decomposed into a set of communicating objects.
2. **A data flow model**
   - The system decomposed into functional modules which accept input data and transform it, in some way, to output data. This is also called a pipeline approach.

4.3 PROCEDURAL DESIGN USING STRUCTURED METHODS

Procedural design using structured methods

- For the component-level design for conventional software components has proposed a set of constrained logical constructs from which any program could be formed.
- Each construct had a predictable logical structure.
- Structured programming is a design technique that constraints logic flow to three constructs that are sequence, construct and repetition.

Procedural design using structured methods

- The structured constructs reduces program complexity and enhances readability, testability and maintainability.
- These allow reader to recognize procedural elements of a module rather than reading the design or code line by line.

4.4 USER INTERFACE DESIGN

User Interface Design

Good user interface design is critical to the success of a system. An interface that is difficult to use will, at best, result in a high level of user errors. At worst, users will simply refuse to use the software system irrespective of its functionality.

If information is presented in a confusing or misleading way, users may misunderstand the meaning of information. They may initiate a sequence of actions that corrupt data or even cause catastrophic system failure.

The system should assist the user providing help facilities and should guide the user in the case of occurrence of an error.

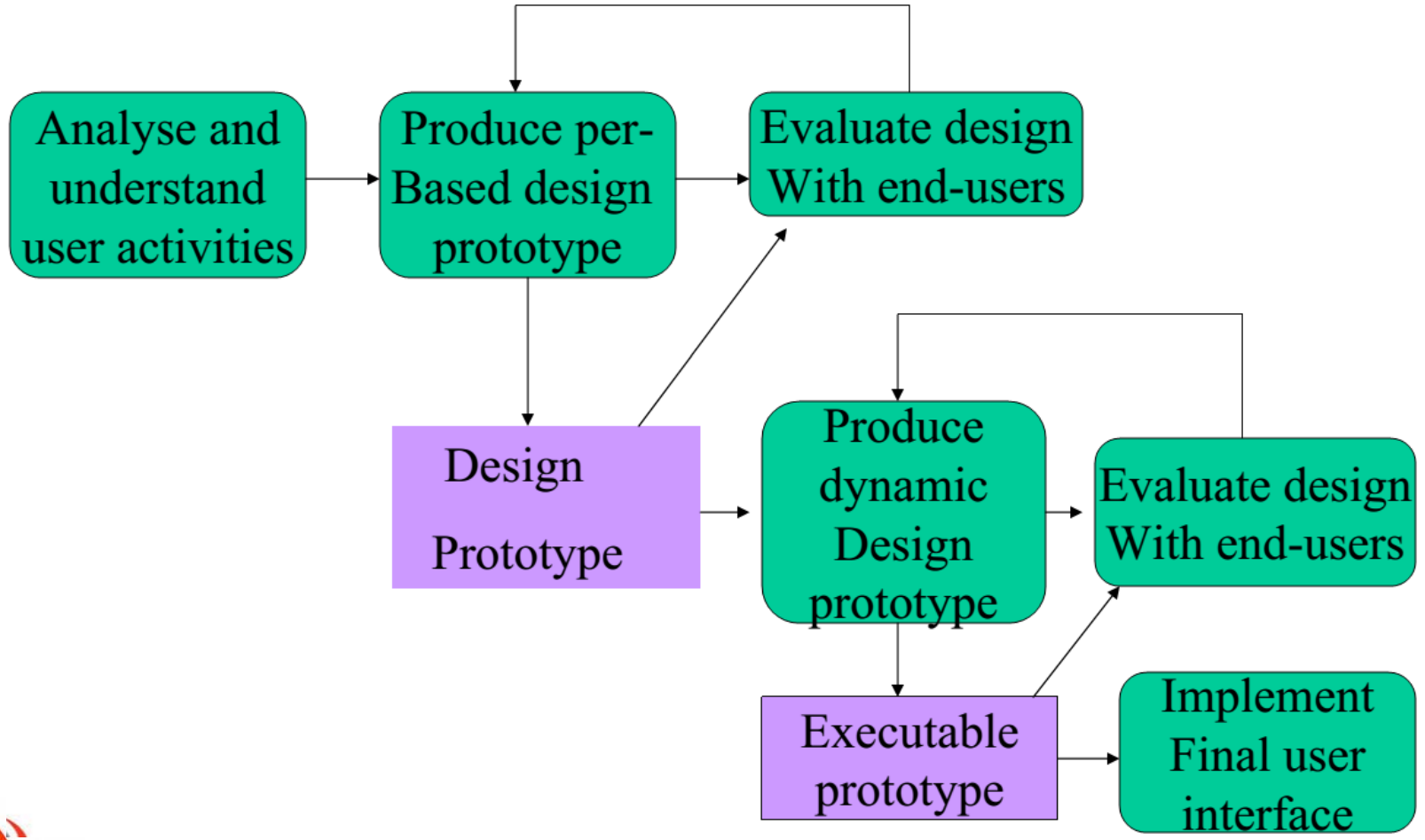
Graphical User Interfaces 

- Although text based interfaces are still widely used, especially in legacy systems, computer users now expect application systems to have some form of graphical user interface.
- **The advantages of GUI are:**
  1. They are relatively easy to learn and use. Users with no computing experience can learn to use the interface after a brief training session.
  2. The users have multiple screens (windows) for system interaction. Switching from one task to another is possible without losing sight of information generated during the first task.
  3. Fast, full-screen interaction is possible with immediate access to anywhere on the screen.

Graphical User Interfaces - characteristics

- **Windows**
  - Multiple windows allow different information to be displayed simultaneously on the user's screen
- **Icons**
  - Icons represent different types of information. On some system icons represents files, on others, icon represents 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.

The user interface design process



Important Design Considerations for Interfaces

- **Response time**
  - too long and too short response times are unacceptable. Constancy in response time helps the user to learn a pace for the interaction with the computer
- **Help**
  - help may be provided as either electronic manuals (add on help) or integrated help (Microsoft office assistant)
- **Error handling**
  - The error, cause and possible solutions should explain in simple terms.

Important Design Considerations for Interfaces

- **Adaptability**
  - with growing familiarity users may migrate from one user class to another
- **Familiarity**
  - The interface should use terms and concepts which are drawn from the experience of the users
- **Consistency**
  - Activating similar operations in the same way
- **Options**
  - Alternative ways of doing things

User interface design principles

- **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.
- **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 type of system user.

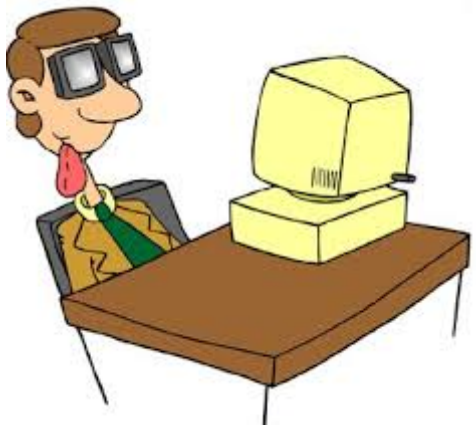
Color in interface design

Some guidelines for effective use of color in user interfaces.

1. You should not use more than four or five separate colors in a window and no more than seven in a system interface. Colors should be used selectively and consistently.
2. Use color change to show a change in system status. If the display changes color, this should mean that a significant event has occurred.
3. Use color coding to support the task which users are trying to perform. If they have to identify anomalous instances, highlight these instances.
4. Be careful about color pairing. Some color combinations are not good for the eye. (eg. Red and Blue)
5. Use color coding in a useful and consistent way. If one part of a system displays error messages in red, then red should not be used for anything else.

Human computer Interaction

- This can be seen when a user enter data into a computer system.
- There are number of approaches that enhance user interaction.



Human computer Interaction

- **Direct interaction**
  - Involve a pointing device. User interact directly with objects on the screen.

    Eg: To delete a file, user may drag it directly to the trash can using mouse.
- **Menu selection**
  - User has to select an option from a list of possibilities. It is often the case that another screen object is selected at the same time and command operates on that object. In this approach to delete a file, user select the file menu and then select the delete command from the list.
- **Form fill**
  - User fills a form that includes fields and buttons. Some fields have associated menus. Buttons initiate some actions on press.

Human computer Interaction

- **Command language**
  - User give necessary commands and parameters to instruct the computer to perform a specific activity.

    Eg: To delete a file, user issues delete command with the file name as a parameter.

- **Natural language**
  - User issues commands in natural language. To delete a file, user may type ”Delete file name xxx”.

The above styles can be mixed in an application.

Interaction Styles - Advantages and Disadvantages

Interaction Style	Main Advantage	Main Disadvantage	Application example
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 and CAD system
Menu selection	Avoid user error. Required little typing.	Show for experienced user. Can be complex if many menu options.	-Most general purpose systems
Form fill-in	Simple data entry. Easy to learn.	Takes up lot of screen space.	-Stock control -Personal loan processing

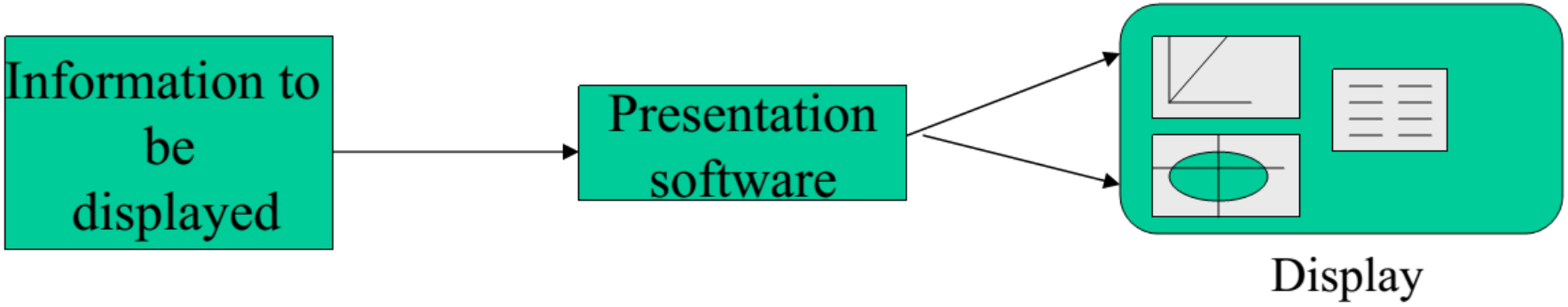
Interaction Styles - Advantages and Disadvantages

Interaction Style	Main Advantage	Main Disadvantage	Application example
Command language	Powerful and flexible.	Hard to learn. Poor error management.	-Operating systems -Library information retrieval systems.
Natural language	Accessible to casual users. Easily expanded.	Requiring more typing. Natural language understanding systems are unreliable.	-Time table systems -www information retrieval systems

Information Presentation

- All interactive systems have to provide some way of presenting information to users.
- The information presentation may simply be a direct representation of the input information or it may present the information graphically.
- It is a good system design practice to keep software required for information presentation separate from the information itself.
- To some extend, this contradicts object-oriented philosophy which suggest that operation on data should be defined with the data itself.

Information Presentation



Interface Evaluation

- Interface evaluation is the process of accessing the usability of an interface and checking that it meets user requirements.
- Systematic evaluation of a user interface design can be as expensive process involving cognitive scientist and graphics designers.
- An evaluation should be conducted against a usability specification based on usability attributes as shown below.

Interface Evaluation

Attribute	Description
Learn ability	How long does it take a new user to become productive with the system?
Speed of operation	How well does the system response to match the user's work practices?
Robustness	How tolerant is the system of user error?
Recoverability	How good is the system at recovering from user errors?
Adaptability	How closely is the system tied to a single middle of work?

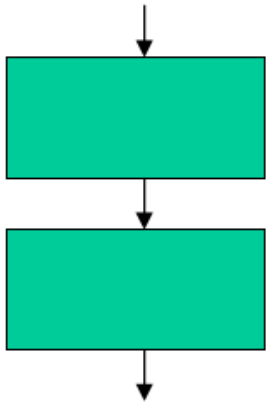
4.5 DESIGN NOTATIONS

Design Notations

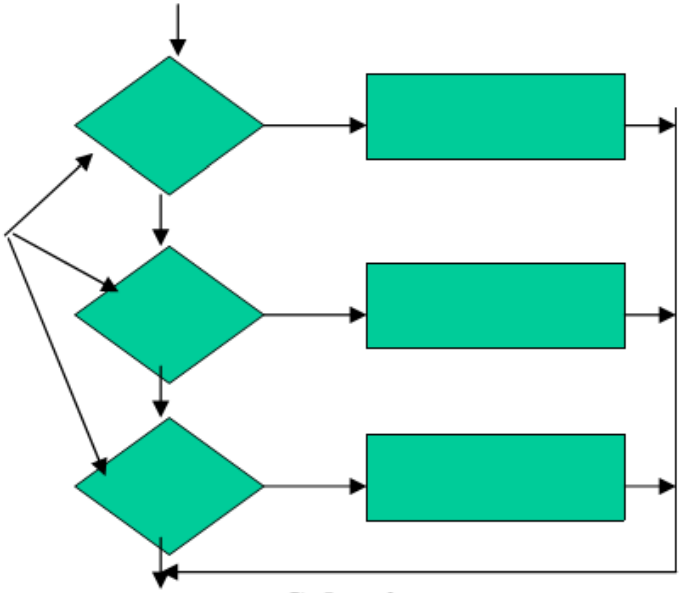
- **Graphical design notation**
  - The activity diagram allows a designer to represent all elements of structured programming.
  - Activity diagram is the descendent of flowchart.

Design Notations

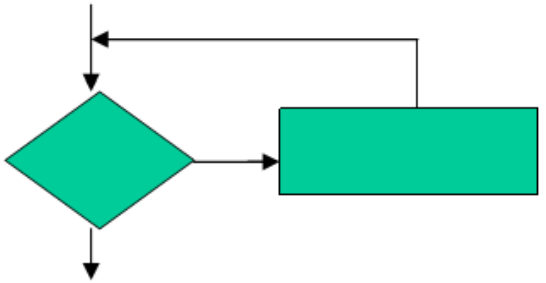
- **Graphical design notation**
  - Flow chart is simple pictorially



Sequence



Selection



Repetition

Tabular Design Notation

- Decision tables provides a notation to translate actions and conditions into a tabular form.
- Following steps are applied to develop a decision table.
  1. List all actions that can be associated with a specific procedure.
  2. List all conditions during execution of the procedure.
  3. Associate specific sets of conditions with specific actions, eliminating impossible combinations of conditions.

Tabular Design Notation

- Define rules by indicating what action(s) occurs for a set of conditions.

Conditions	1	2	3	4	5	6
Regular customer	T	T				
Silver customer			T	T		
Gold customer					T	T
Special discount	F	T	F	T	F	T
Actions						
No discount	✓					
Apply 8% discount			✓	✓		
Apply 15% discount					✓	✓
Apply additional x% discount		✓		✓		✓

Resultant Decision Table

Program Design Language

- This is also known as structured English or pscudo code.
- This uses vocabulary of one language (English) and overall syntax (structured programming language) of another.
- PDL actually is not a programming language due to the use of narrative text.
- There are tools that can translate PDL into programming language.

Program design language has been found as the best design notation.