UNIT–3 : Object Oriented Analysis and Design (PYQ Solution)

AKTU · Object Oriented System Design with C++ (BCS054) · Solutions by **Chirag Jain**, Ycotes

SECTION A Answers (≈70 Words Each)

Q1. What is encapsulation in object-oriented design? (Jan 2026)

Encapsulation is the packaging of data (attributes) and the operations that act on them into a single unit, the class, while hiding internal details behind a public interface. Access specifiers (`private`, `protected`, `public`) control visibility. It gives data security, modularity and lets the implementation change without affecting users of the class.

Q2. How are classes translated into data structures? (Jan 2026)

In the design phase a class is mapped to a language construct: its attributes become fields of a record/struct and its operations become functions that take the object (or its pointer) as a parameter. In non-OO languages like C: `struct Student {int roll; char name[20];}` plus `void display(struct Student *s)`. Inheritance is mapped by embedding the parent struct as the first member, and associations by pointers or arrays of pointers. Methods are stored once per class, data once per object.

Q3. What is a Candidate key? (2022)

A candidate key is a minimal set of attributes that can uniquely identify every record (object) in a table or class; it has no redundant attribute. A relation may have several candidate keys, from which one is chosen as the primary key. Example: in `Student(roll, email, name)`, both `roll` and `email` are candidate keys. In object design they help to identify and map objects to tables.

Q4. How is it different from multiple inheritance and modelled by using nested generalization? (2021)

Multiple inheritance lets a class have more than one direct superclass, causing ambiguity (diamond problem). Nested generalization arranges classes in a hierarchy of several levels, each with a single parent, so the same features are obtained through chained single inheritance, e.g. `Vehicle → MotorVehicle → Car`. It avoids conflicts, is simpler to implement and is supported by languages without multiple inheritance.

Q5. Define and differentiate Procedural and OOP with example. (2021)

Procedural programming organises a program as a set of procedures acting on shared data; OOP organises it as objects combining data and methods.

Procedural	OOP
Top-down, function-centred	Bottom-up, object-centred
Data is global and exposed	Data is hidden (encapsulated)
No inheritance or polymorphism	Supports both
Less reuse	High reuse
Example: C	Example: C++, Java

SECTION B Answers (≈250 Words Each)

Q1. Discuss abstraction and encapsulation with suitable examples. Explain reusability, extensibility, and robustness in object-oriented programming. (Jan 2026)

Abstraction means exposing only the essential features of an object and hiding irrelevant detail. It defines the *what*. In C++ it is achieved with classes, abstract classes and interfaces. Example: a `Stack` offers `push()` and `pop()`; the user does not know whether an array or list is inside.

Encapsulation binds data and functions into a class and restricts direct access using `private` members. It is the *how* of protection.

```
class Account {
    double balance;                 // hidden
public:
    void deposit(double a) { if (a > 0) balance += a; }
    double get() { return balance; }
};
```

Quality factors

- **Reusability:** classes written once can be used in many programs through inheritance, composition and templates. A well-tested `Date` class is reused in payroll, library and billing systems; this saves time and cost.
- **Extensibility:** new features are added with little change to existing code, by deriving new classes or overriding virtual functions. Adding `Triangle` to a `Shape` hierarchy needs no change to the code that draws shapes (open–closed principle).
- **Robustness:** the ability to handle invalid input and abnormal conditions without failing. Encapsulation validates data (`deposit` rejects negative amounts), and C++ exceptions (`try/catch`) handle run-time errors gracefully.

Quality	Supported by
Reusability	Inheritance, templates, composition
Extensibility	Polymorphism, abstract classes
Robustness	Encapsulation, exception handling, constructors

In conclusion, abstraction and encapsulation make classes simple and safe, and together with reusability, extensibility and robustness produce high-quality object-oriented software.

Q2. Prepare a DFD for computing the volume and surface area of a cone. Inputs are height and the radius of the base. Outputs are volume and surface area. Discuss some ways of specifying operations. (2021)

Formulas: slant height $l = \sqrt{(r^2 + h^2)}$; volume $V = (1/3)\cdot\pi\cdot r^2\cdot h$; total surface area $A = \pi\cdot r\cdot(r + l)$ (base πr^2 plus curved $\pi r l$).

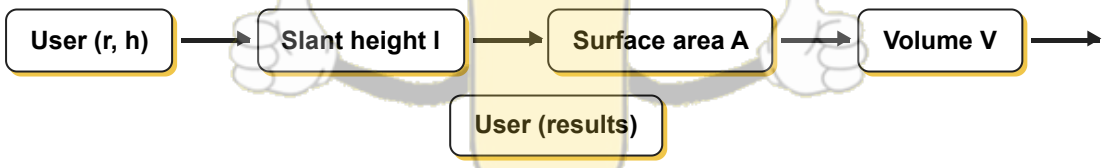
Processes

1. **Compute slant height:** input r, h → output l.
2. **Compute volume:** input r, h → output V.
3. **Compute surface area:** input r, l → output A.

Level-1 DFD

```
User --r,h--> [1 Slant height] --l--> [3 Surface area] --A--> User
User --r,h--> [2 Volume]  --V--> User
```

External entity: User (source of height and radius; sink of results). There are no data stores because nothing is retained.



DFD for cone volume and surface area

Ways of specifying operations (process specifications)

- **Structured English / pseudocode:** `l = sqrt(r*r + h*h); V = pi*r*r*h/3`.
- **Decision tables** and **decision trees** for conditional rules.
- **Mathematical formulas / equations** (as above).
- **Pre- and post-conditions:** precondition r > 0, h > 0; postcondition $V = \pi r^2 h/3$.
- **Flowcharts** or **state/transition tables**.
- Plain **natural-language** description (least precise).

Example check: r = 3, h = 4 → l = 5, $V = (1/3)(\pi)(9)(4) = 12\pi \approx 37.7$, $A = \pi\cdot 3\cdot(3 + 5) = 24\pi \approx 75.4$.

SECTION C Answers (≈500 Words Each)

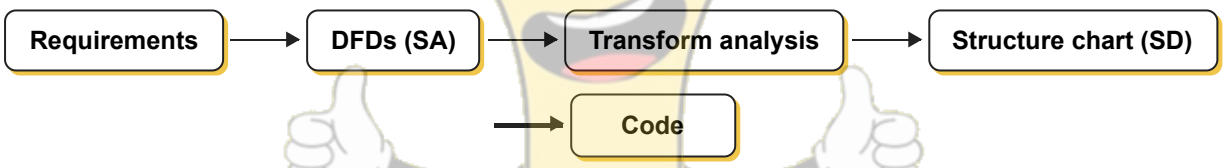
Q1. Describe structured analysis (SA), structured design (SD) and Jackson Structured Development (JSD) in detail. (Jan 2026, 2023)

Structured Analysis (SA) studies the problem and produces a functional model of the system using **top-down decomposition**.

- Tools: **Data Flow Diagrams** (processes, data flows, data stores, external entities), **data dictionary**, **process specifications** (structured English, decision tables), **entity-relationship diagram**, **state transition diagram**.
- Steps: draw context diagram (level 0), decompose into level 1, 2 … DFDs until processes are simple, then define the data dictionary.

Structured Design (SD) converts the DFDs into a **hierarchy of modules** shown in a **structure chart**.

- Uses **transform analysis** (input, central transform, output branches) and **transaction analysis** (a dispatching module choosing one of many paths).
- Aims at **high cohesion and low coupling**; the chart shows module calls and data/control couples.



SA/SD flow

Jackson Structured Development (JSD) models the **real-world entities and their time-ordered actions** before functions are decided. It has these steps:

1. **Entity-action step:** list entities and actions (events) they perform or suffer.
2. **Entity structure step:** draw Jackson structure diagrams using **sequence, selection and iteration** for the life history of each entity.
3. **Initial model step:** model the real world as long-running processes connected by messages or state vectors.
4. **Function step:** add functions (outputs, reports) to the model.
5. **System timing step:** decide scheduling and synchronisation.
6. **Implementation step:** map processes onto available processors, using inversion and scheduling.

Aspect	SA/SD	JSD
Starting point	Functions and data flow	Entities and their actions
Focus	Process decomposition	Time-ordered behaviour
Diagram	DFD, structure chart	Entity structure diagram, network diagram
Suitable for	Data-processing systems	Real-time and event-driven systems

Example: for a library, SA draws processes Issue Book and Return Book on a DFD; JSD models entities Book and Member with life histories (acquire, issue, return, discard).

In conclusion, SA/SD is function-oriented, while JSD is entity-oriented; both are non-object-oriented predecessors that influenced later object-oriented methods.

Q2. Discuss object representation, physical packaging, and documentation design considerations. Explain how object-oriented concepts are mapped to non-object-oriented languages. (Jan 2026)

Object representation (design of data structures)

- Each object is stored as a record of its attributes; methods are shared per class.
- Attributes may be primitive values, pointers to other objects (for associations), or embedded objects (composition).
- Choices consider memory, speed and access: arrays for fixed collections, linked lists for dynamic ones.
- Derived attributes may be stored (faster) or computed (consistent).

Physical packaging

- Group related classes into **modules/packages/files** (header and implementation) to hide implementation.
- Separate **interface (.h)** from **implementation (.cpp)**.
- Aim for high cohesion within a package and low coupling between packages; minimise cyclic dependencies.
- Package decisions also consider team organisation and compilation time.

Documentation

- Design documents must record class specifications, responsibilities, diagrams, assumptions and decisions.

- Code comments and API documentation (preconditions, postconditions, parameters).
- Keep documents consistent with the code.

Mapping OO concepts to non-OO languages (e.g. C, Pascal)

OO concept	Mapping
Class	`struct` plus a set of functions
Object	A variable of that struct
Method	Function taking object pointer as first parameter
Inheritance	Parent struct embedded as the first field
Polymorphism	Table of **function pointers** in the object (like a vtable)
Encapsulation	Keep struct definition in .c file, expose opaque pointer and functions
Association	Pointers or arrays of pointers
Message passing	Function call or indirect call through pointer
Object identity	Address or generated ID

```
struct Shape { double (*area)(struct Shape*); };
struct Circle { struct Shape base; double r; };
```

In conclusion, careful representation, packaging and documentation keep a design implementable, and OO designs can still be coded in non-OO languages by disciplined use of structs and function pointers.

Q3. Differentiate: (i) SA/SD and OMT, (ii) SA/SD and JSD. (2021)

(i) SA/SD versus OMT (Object Modeling Technique)

Point	SA/SD	OMT
Approach	Function-oriented, top-down	Object-oriented
Primary model	DFD (functional)	Object model (class diagram)
Other models	Data dictionary, ER	Dynamic model (state), functional model (DFD)
Unit	Process / module	Object / class
Data and function	Separated	Combined
Reuse and change	Poor, changes ripple	Good, local
Phases	Analysis, design	Analysis, system design, object design, implementation

(ii) SA/SD versus JSD

Point	SA/SD	JSD
Starting point	Required functions	Real-world entities and actions
Main diagram	DFD and structure chart	Entity structure diagram, system specification network
Modelling	Function decomposition	Time-ordered life histories
Concurrency	Not central	Core: long-running processes
Best for	Data-processing	Real-time, event-driven
Functions	Considered first	Added late (function step)

Summary: SA/SD decomposes by *what the system does*; JSD models *the world the system serves* first; OMT models *objects with data and behaviour together*. OMT is the most suitable for modern, changing systems.
