**Key Topics:**

1. **Finite State Machine**
2. **UML Statechart Diagrams**
3. **Components of State Diagrams**
4. **Statechart Diagrams and Notations**
5. **Dynamic Modeling Examples**
6. **Practical Tips for Dynamic Modeling**
7. **State Chart Diagram vs. Sequence Diagram**
8. **When to Use State Diagrams**
9. **Requirements Analysis Review**
10. **When is a Model Dominant?**

**1. Finite State Machine**

**Concept**:

* A finite state machine is a model used to represent and control execution flow. It consists of states, transitions, and events.
* **Example**: A light switch
  + **States**: Light on, Light off
  + **Transitions**: Flip switch on, Flip switch off

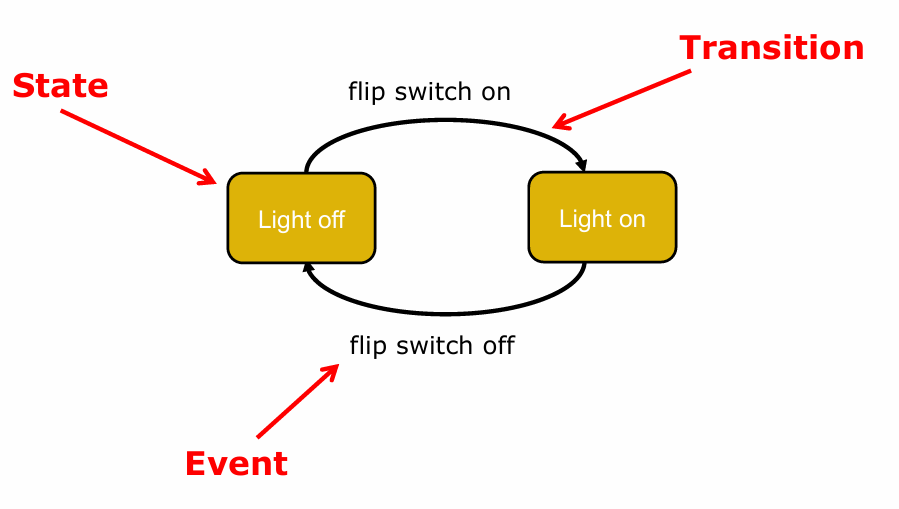
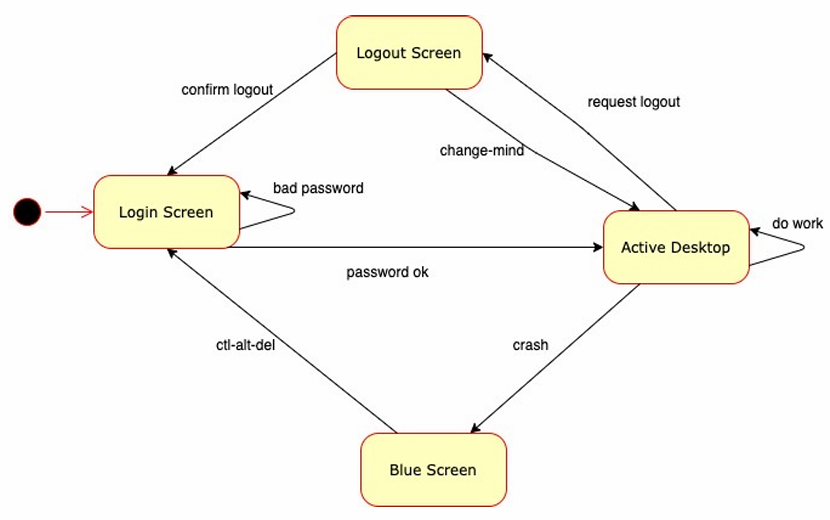
**2. UML Statechart Diagrams 100% exam**

**Definition**:

* UML state diagrams describe the sequence of states an object goes through in response to external events.
* They are extensions of the finite state machine model.
* how individual objects change their state in response to events.

**Key Concepts**:

* **State**: A condition satisfied by the attributes of an object.
* **Transition**: A move from one state to another.
* **Event**: A potential trigger for a state change.

State: Light off

Event: flip switch on

Transition: Light off -> Light on

State: Light on

Event: flip switch off

Transition: Light on -> Light off

**3. Components of State Diagrams**

**States**:

* Capture conditions that hold for a period.
  + Example: Light is on, Light is off.
* abstraction of the attributes of a class with duration

**Transitions**:

* Labeled by events, indicating a change from one state to another.

**Events**:

* Something that happens at a point in time (e.g., button press, mouse click).
* Can trigger internal (no state change) or external transitions (state change).

**Actions**:

* May be executed because of an event.

**Statechart Example**: Interaction with a PC

* **States**: Blue screen, Login screen, Active desktop, Logout screen.
* **Events**: ctrl-alt-del, username-passwd, request-logout, confirm-logout.

**4. Statechart Diagrams and Notations**

**UML Statechart Diagram Notation**:

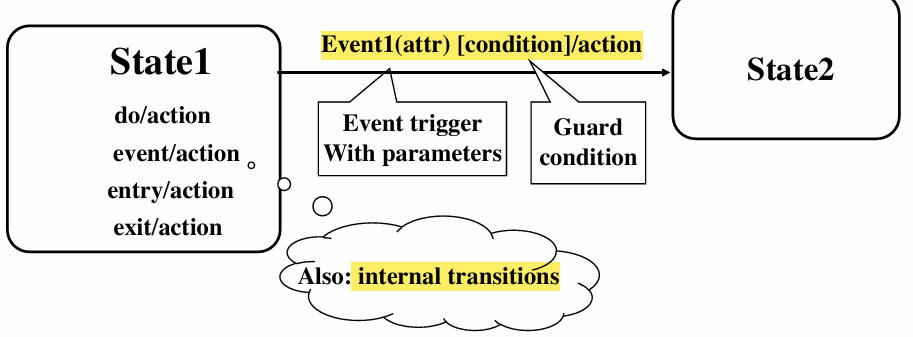
* Based on statecharts by Harel (1987) with object-oriented modifications.
* Can be mapped into a finite state machine (FSM).

**Elements**:

* **State**: Represented by a rounded rectangle.
* **Transition**: Arrow indicating a state change.
* **Event/Action**: Labelled on transitions to show the event triggering the transition and the action performed.

**Example Notation**:

State1 --[event(attr) [condition]/action]--> State2



* **Entry/Exit Actions**: Defined with entry/action and exit/action.
* **Do Actions**: Represent ongoing activities within a state (do/action).

**Example**:

Idle --coins\_in(amount)/set\_balance--> Collect Money

Collect Money --cancel/refund\_coins--> Idle

Collect Money --[select(item)]/do:dispense\_item--> Dispense Item

**5. Dynamic Modelling Examples**

**Example 1: Vending Machine**:

* **States**: Idle, Collect Money, Dispense Item
* **Transitions**: coins\_in(amount), cancel, select(item), dispense\_item
* **Events and Actions**:
  + coins\_in(amount) / add to balance
  + do: test item and compute change
  + do: make change
  + do: dispense item

**Example 2: Incident Management**:

* **States**: Incident Documented, Incident Handled, Incident Archived
* **Events and Transitions**: Based on the state of the incident object in a system like FRIEND.

**Example 3: Toy Car Control**:

* **States**: Stationary, Forward, Backward, Headlight On, Headlight Off
* **Transitions**: power on, power off
* **Events**:
  + power on: Car moves forward, headlight shines
  + power off: Car stops, headlight goes out

**Dynamic Model for Toy Car**:

State: Stationary

Event: power on

Transition: Stationary -> Forward

State: Forward

Event: power off

Transition: Forward -> Stationary

**6. Practical Tips for Dynamic Modelling**

* Construct dynamic models only for classes with significant dynamic behaviour to avoid analysis paralysis.
* Consider only relevant attributes and use abstraction if necessary.
* Look at the application’s granularity when deciding on actions and activities.
* Reduce notational clutter for clarity.

**7. State Chart Diagram vs. Sequence Diagram**

**State Diagrams**:

* Help identify changes to objects over time.

**Sequence Diagrams**:

* Help identify temporal relationships between objects over time.
* Show the sequence of operations in response to events.

**Comparison**:

* Use state diagrams for understanding the behaviour of a single object across multiple use cases.
* Use sequence diagrams for understanding the collaboration among multiple objects within a use case.

**8. When to Use State Diagrams**

* Useful for describing the behaviour of an object across several use cases.
* Not suitable for describing behaviour involving multiple objects collaborating.
* Combine with other techniques for a comprehensive view.
* Use only for classes exhibiting interesting behaviour to avoid unnecessary complexity.
* Particularly useful for User Interface and Control objects.

**Reference**: Martin Fowler, UML Distilled

**9. Requirements Analysis Review**

**Steps**:

1. **Transformations**:
   * Create scenarios and use case diagrams by talking to clients, observing, and conducting thought experiments.
2. **Structure**:
   * Create object and class diagrams.
   * Identify objects, associations, multiplicity, attributes, and operations.

**Models**:

* **Functional Modelling**: Focus on use cases and scenarios.
* **Object Modelling**: Focus on class and object diagrams.

**Dynamic Modelling**:

* Create state diagrams for dynamically interesting objects.
* Create sequence diagrams to identify senders and receivers of events, sequence of events, event dependencies, and concurrency.

**10. When is a Model Dominant?**

**Functional Model**:

* Dominant when the model performs complicated transformations or computations.

**Object Model**:

* Dominant when the system has non-trivial data structures.

**Dynamic Model**:

* Dominant when the model involves many types of events, including inputs, outputs, exceptions, and errors.

**Examples**:

* **Compiler**: Functional model is most important due to complex transformations.
* **Database Systems**: Object model is most important due to focus on data structures.
* **Spreadsheet Program**: Functional model is crucial for computations; dynamic model is also important for event handling.

# Part B

**2. System Design Process**

**Overview**:

* Software Design focuses on the solution domain (implementation), while requirements analysis focuses on the problem domain.
* creative process, goal driven, solving some problem constraint driven, good designs = simple, coherent(连贯), adequately meets requirements, adaptable(适应性强)
* In system design, objects identified during analysis are grouped into subsystems.
* Purpose: Bridge the gap between the system requirements and a system implementation

**Process Steps**:

* Define design goals.
* Decompose the system into subsystems.
* Select off-the-shelf and legacy components.
* Map subsystems to hardware.
* Choose a data management infrastructure.
* Decide on access control policies.
* Select a global control flow mechanism.
* Handle boundary conditions.

**3. Modularity模块性**

**Definition**:

* Modularity is essential for making software intellectually manageable.
* Monolithic software is hard to understand due to numerous control paths, variables, and overall complexity.

**Principle**:

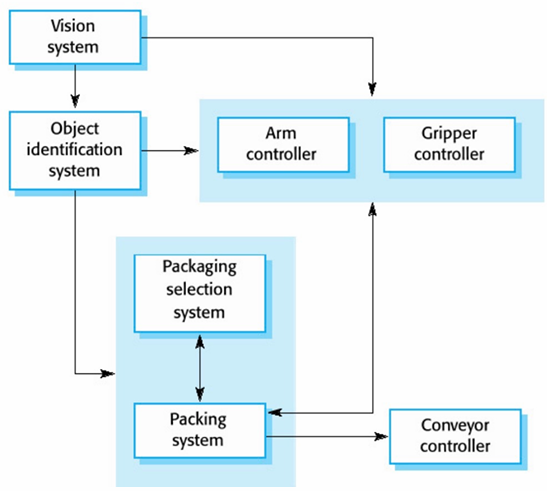
* Break the design into many modules to reduce complexity and cost.

**Example**:

* Decomposition of a robotic packing system in a factory.

**Diagrammatic Decomposition**:

* Use simple block diagrams (box and line diagrams) to show entities and relationships.
* Useful for communication with stakeholders and project planning.



**4. Information Hiding**

**Concept**:

* Every module should hide aspects of its implementation, exposing only an understandable interface.

**Benefits**:

* Reduces likelihood of side effects.
* Limits global impact of local design decisions.
* Emphasizes communication through controlled interfaces.
* Discourages the use of global data.
* Leads to encapsulation, resulting in higher quality software.

**5. Identifying Subsystems**

**Process**:

* Break down the solution domain into simpler parts.
* A subsystem is a collection of classes, associations, operations, events, and constraints.
* The decomposition process involves backtracking, evaluation, and revision.
  + important to get the decomposition right
  + subsystems implemented by different teams
  + bad decomposition can lead to unworkable designs

**Criteria**:

* Primary: What kind of service is provided by the subsystems?
* Secondary: Can the subsystems be hierarchically ordered (layers)?
* Most interaction should be within a subsystem rather than across boundaries.

**6. Modular Design**

**Definition**:

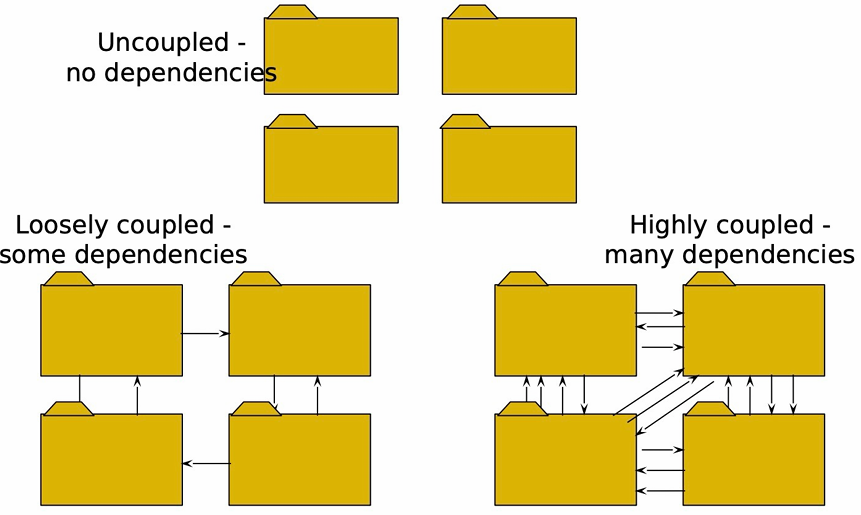
* A design is modular when each activity is performed by exactly one component, with well-defined inputs and outputs.

**Goal**:

* Minimize the impact of later changes by abstracting from implementation details.

**!!!!!!! 7. Coupling耦合 and Cohesion 内聚**

**Definitions**:

* **Cohesion**: Dependence among classes within a subsystem.
  + High cohesion: Classes perform similar tasks and are related (via associations).
  + Low cohesion: Miscellaneous classes with no associations.
* **Coupling**: Dependencies between subsystems.
* 
  + High coupling: Changes in one subsystem affect others.
  + Low coupling: Changes in one subsystem do not affect others.

**Goal**:

* Achieve maximum cohesion and minimum coupling. Classes高关联，低相关影响

**Coupling Levels**:

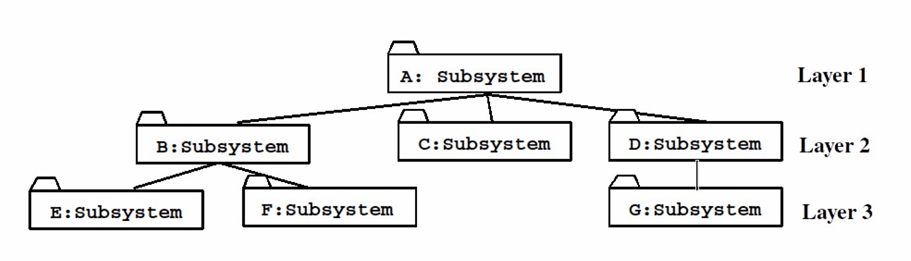
* **Content Coupling**: One module modifies or relies on the internal workings of another.
* **Common Coupling**: Modules share global data.
* **Control Coupling**: One module controls another’s logic.
* **Stamp Coupling**: Modules share a composite data structure.
* **Data Coupling**: Modules share data through parameters.
* **Uncoupled**: No data is shared.

**Cohesion Levels**:

* **Functional Cohesion**: All parts contribute to a single task.
* **Sequential Cohesion**: Output from one part is input to another.
* **Communicational Cohesion**: Parts operate on the same data.
* **Procedural Cohesion**: Parts follow a specific sequence.
* **Temporal Cohesion**: Parts are grouped by timing.
* **Logical Cohesion**: Parts do similar things logically.
* **Coincidental Cohesion**: Parts are grouped arbitrarily.

**8. Partitions and Layering**

Partitioning and layering are techniques to achieve low coupling.

****

**Definitions**:

* **Partitions**: Divide a system into independent subsystems at the same level of abstraction.
* **Layers**: Subsystems that provide services to higher layers and depend on lower layers.

**Benefits**:

* Supports incremental development of subsystems.
* Limits the impact of changes to layer interfaces.

**Relationships**:

* **Layer A “Calls” Layer B**: Runtime dependency.
* **Layer A “Depends on” Layer B**: Compile-time dependency.
* **Partition A “Calls” Partition B and vice versa**: Mutual knowledge but not deep.