Software Engineering

- software **process**
    - **Specify**
        - software **requirement and specification**
            - contained required **establishing procedures** and **constraints** and **description of services**
            - **requirement**
                - types
                    - users requirement (understandable)

                      natural language (describe functional and nonfunctional requirements) + diagrams & tables + operational constraints → for customers (better understand and point out problems)
                        - natural language problems
                            - Lack of clarity
                            - Requirements confusion
                            - Requirements amalgamation
                    - system requirement

                      detailed descriptions of services → **contract** between clients and contractor
                    - software requirement

                      detailed software description → for developers
                    - **functional and non functional requirement**
                        - **functional requirements**

                          describe what the **system should do (provide + behavior)** how system react to inputs
                        - **non-functional requirements more critical**

                          describe how the **system works**, some **constraints and properties** (E.G: reliability,response time and storage requirements)
                            - Non-Functional Classifications
                                - product requirements

                                  delivered product must **behave in a particular way** e.g. execution speed, reliability,security
                                - Organisational requirements

                                  organisational **policies and procedures** e.g. process standards used, implementation requirements (**java** as programming language)
                                - External requirements

external to the system and its development process e.g. interoperability requirements, **legislative requirements**

- Performance requirements

  respond to some requests in less that 1 second ...

- **Domain requirements**

- Good requirements should be **complete and consistent**
- requirement documents

  requirements involve many stakeholders, we need to elicit and analyze them.
  multiple stakeholders with different requirements

  - requirements come from (viewpoint)
    - other systems

      Copying/modifying
    - legacy system

      Copying and fixing
    - competitors
    - prototyping

      a version (user interface)
    - viewpoints

      may comes from
      1. providers and receivers
      2. Systems that interact directly with the system being specified
      3. Regulations and standards
      4. Sources of business and non-functional requirements.
      5. Engineers (develop and maintain)
      6. Marketing and other business viewpoints.
    - interviewing

      should not have pre-conceived ideas.
      - Closed interviews

        do not need to change, just go through the questions (pre-defined)
      - Open interviews

        impromptu (no pre-defined)
    - Ethnography

      Combines ethnography with prototyping
  - **requirements Engineering Processes**

    The requirements elicitation and analysis stage is iterative and involves
    domain understanding,
    requirements collection,
    classification,
    structuring,
    prioritisation,
    validation.

- feasibility study

  worthwhile, achievable (budget), integration (with other systems)
- **requirements elicitation (iterative)**

  what user require (and some constraints)
  - find out the services
  - application domain
  - system’s operational constraints.
- **Requirements analysis (iterative)**

  classify, prioritise and negotiate
  stakeholders' requirements always get contradictions
- Requirements validation

  match users' requirements (may be use a prototype)
- Requirements management

  changes

- security requirements (6)
  - confidentiality

    keep stuff private, people can not see something (code) they are not allowed to.
    - Encryption (hard security)

      better
    - Permissions (soft security)

      if the hard disk has been removed from the system right directly, doesn't matter what permission you have on the files, they can be read
  - integrity

    be able to see/detect when somebody modified the data
    - Integrity approaches
      - ~~*CRC Checking*~~

        useless after recomputing
      - ~~*Hash value over data*~~

        useless after recomputing
      - Hash value over data + secret value

        generate hash value, but use secret value when doing computation
        me and receiver have secret value (only we 2 know), hash value is generated using my secret value send to the other side, and receiver use his secret value to do the same computation. without secret values, somebody can't change the date and recompute the hash value
        but we have to share the secret value first (difficult)
      - Hash value encrypted using asymmetric cipher

        encrypt the data use receiver 's public key, and receiver can use public key to de-encrypt data
  - Authentication and Authorization

proof who are you and set what you are allowed to do

- Authentication

  Who are you?

- Authorization

  what you are allowed to do ?

- techniques

  Usernames,Passwords, hardware (cards, dongles), Biometrics, ask user to send messages back......

- Non-repudiation

  不可抵赖

  - in practice

- Availability (Performance security)

  use 9s terminology

  - 9s terminology

    how many of the time the system is expected to be out (down)
    the more 9s means more available of the system (99.9999%)

  - in practice

    need to specify:
    Worst case scenarios
    Worst case delay as well as down time
    How the system can degrade gracefully

- some services that support security

  - log

    - standard log

      记录登录登出，遇到attacker攻击的时候可能能够确定攻击者的IP...

    - Failed login log

    - Unusual activity log

      大额交易，重复交易

    - Alert log

  - Bell–LaPadula model (in the exam)

    a approach to security based on security clearance
    "向上写入，向下读取 write up, read down"（WURD）
    stop people unwaitingly releasing security on data, keeps data in security level it requires

    - security clearance level

      Top-Secret(4), Secret(3), Sensitive(2), Unclassified

    - rules

      - no read-up

        if I am level 2, I can not read up to level 4 document (can read level 1 & 2 documents)

- no write-down

  A document cannot be copied/included/saved with another document with a lower security clearance level

- trusted subjects

- specifying security
  - open

    as open as possible
    encryption algorithm always make mistakes, needs to be updated all the time

  - security policy
  - standards compliance

- requirements checking
  - check
    - validity
    - consistency
    - completeness
    - realism
    - verifiability (critical)
  - check method
    - scenarios

      threat
      one use case can have many scenarios
      Help developer clearly see what must happen with their codes in given circumstances

    - agile requirements tool
      - cucumber

        define both requirements and tests of some scenarios in a language format - gherkin
        a scenario related to a test
        formalize this link between requirements and testing

- **specifications**
  - specification operations

    To specify behavior, define the inspector operations for each constructor operation.

    - Constructor operations

      create new
      for example, make new string

    - Inspection operations

      return some information about the objects
      for example, getlength( )

  - Specification Techniques

to structure a specification, an **architectural design** is essential

- **Algebraic approach**
    - introduction

      declare the sort (type name) of the entity
      as well as the import
    - description

      informal
    - signature

      the type of operations can be done within the context
    - axioms

      must be true for the thing to operate (a - a = 0)
- **Model-based approach**

  e.g. Petri net

- **Formal specifications**

  part of the formal methods
  describe the system in terms of some **formal mathematical approach**
  mathematical notation with precisely defined vocabulary, syntax and semantics.
  improve system quality
  discover system problems in requirements and **reduces requirements errors**
    - do many works at the beginning to reduce errors and reduce the amount of rework on the requirements and improve the system
    - critical system usually follows **waterfall model**
    - **clear and unambiguous**
    - interface specifications

      algebraic approach
        - break large system into sub-systems
        - sub-systems can interact with each other without know the internal code
        - Interfaces may be defined as abstract data types or object classes
        - Clear and unambiguous
        - in critical system
            - sector
    - **ASML**

      Abstract State Machine Language

- **Design**
    - **convert specification into executable system**
    - process to realize the specification
    - design **structure**
    - design process

- Architectural design

    1. separate system into **sub-systems**
    2. identified and documented the relationship between subs

- Abstract specification

    specification (services and constraints) of subs

- Interface design

    design and documented **interfaces** between subs

- Component design

    allocate services to subs

- Data structure design
- Algorithm design

- design (structured) Methods

    graphical models

    - **modeling** (7)

        generally simplification of the system
        abstract description of the system
        not the code, but the representation
        clear enough (for structures, constraints , requirements, interfaces .... of the system)
        simplification

        - models perspectives

            - external perspective

                for example: use case modeling
                showing the system's context or environment
                can not see what going on in the system, but use cases describe the function (interaction) of the system

            - behavioural perspectives

                how the system behave internally
                what stimulus it responds to
                how internal stages changes as response to different stimulus

            - structural perspective

                show relationships between main components of the system
                system or data architecture

        - system model advantages and disadvantages

            - advantages

                easier to understand
                clear
                focus on what is important
                representation , abstraction, simplification

            - disadvantages

                1. system models do not provide non-functional requirements
                (when we talked about security,non functional requirements can be as important

as functional requirements)
2. sometimes too detailed and difficult for users to understand

- model type
  - **Data processing model**
  - **Composition model**
  - **Architectural model**

    naming different components , showing which one is connected to which one
    showing relationships between systems
  - **Classification model**
  - **Stimulus/response model**
  - **Context Models**

    illustrate the boundaries of a system
    now we know which things we have to have as interacting with our system and which things are built into our system
    固定了和谁interact
    - use case diagram
    - Architectural model
  - **process model**

    different processes have to be done before the next process can begin
    typically used in manufacturing
    - Data-flow model

      can include human part (human processes) and system part (input and output results will be double checked by human being)
  - **behavioural models**

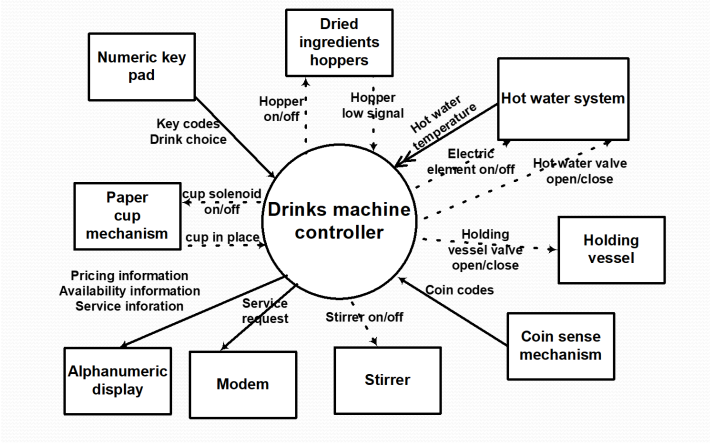
    how the system react to different inputs
    A state transition model showing system states and triggers
    - **Data processing models**

      activities and data flowing in and out of those processes.
      Simple and intuitive notation
      Show end-to-end processing of data
      - **Data flow diagrams** (DFD)

        modeling the system's **data processing**
        showing the **data exchange** between a system and other systems
        can produce a DTD for a non-existing system
        simple and intuitive
        - **top-down process**
        - show a **functional perspective**

          数据流图显示了一个功能透视图，其中每个转换代表一个单一的功能或过程，这在需求分析中特别有用，因为它显示了端到端的处理。
          each block of analysis is a processing unit

- allow the **decomposition of a model into sub-models**
- Example 1
  - DFD

    - how the controller control hardware external to the internal system
      - Jordan notation

        (It was developed moved from what were merely information systems to more real time and embedded systems.So it has some extensions to the notation that allow DFDs to be done for real time systems)

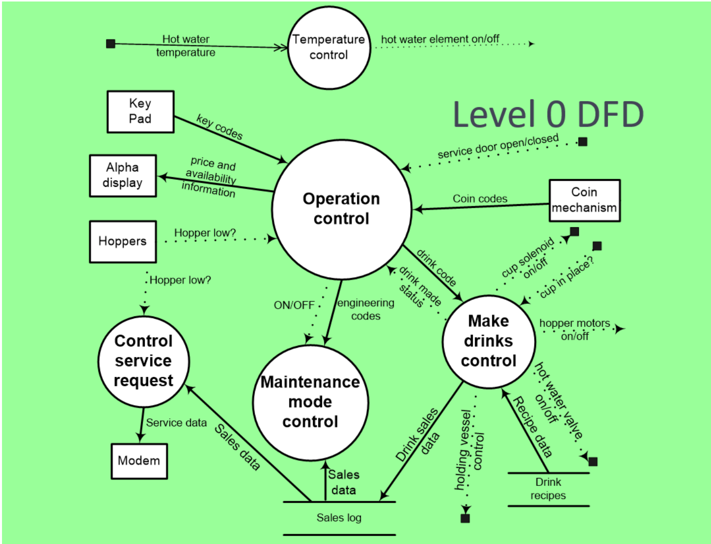
        - dotted line

          implies control signals flow, switch something on and off
        - solid line

          implies data flow
        - double arrow head line

          continuous flow of information
        - single arrow head line

          discrete data flow
  - LEVEL 0 DFD



further deep level

every time we get to a further deep level in a DFD diagram, the process is **more and more simplified**

- LEVEL 1 DFD

- **State machine models**
  - **statechart diagram (part of UML)**

    behaviour of the system **in response to external and internal events (stimulus)**

    often used for modelling **real-time systems**

    show system **states as nodes** and **events as arcs** between these nodes.

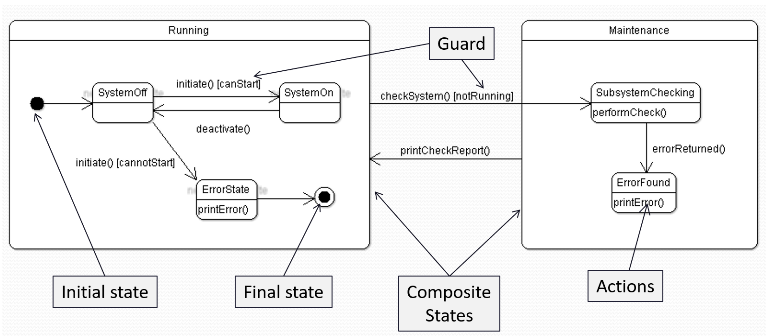
    When an event occurs, the system moves from one state to another

    does not show flow of data within the system

    system goes **from this state to next state** depends on what button been pressed or what events happens

    if the system is complex, allow several states running concurrently

    - allow the **decomposition of a model into sub-models**
    - components



- states

  as nodes

  denoted by Rounded rectangles

  - state name
  - description of action performed
  - sub-diagram

a state can contain a sub-diagram within it (this state is also called a **composite state**)

- events

  label on arcs

  - guard

    marked by [ ], denote a condition must be true for the transition can move forward
    the value of guard (true or false) depends where process will go

- initial state

  denoted by solid circle

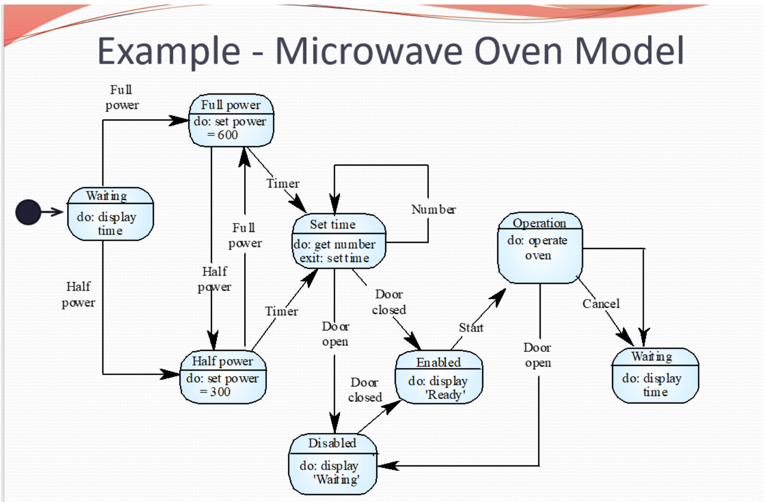
- final state

  denoted by solid circle and a ring

- stimulus table (optional)

  Can be complemented by tables describing the states and the stimulus

- Example 2



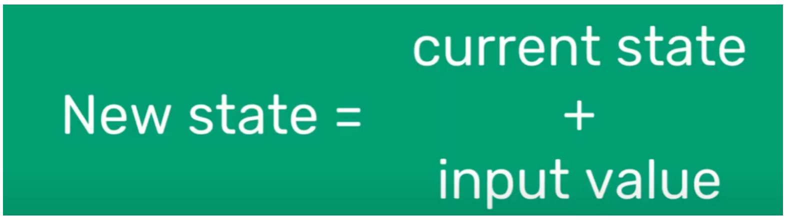
- **All the actions are done after the changes of state,so it shows you as you're moving from state to state.**

  the advantages of putting an action after an event in terms of moving into a change of state is that if you're entering the same state from different locations,obviously you can have different actions, It's a bit more flexible, so by putting it on the arc you can make it relevant to why you're moving into that particular state.

- **Finite State Machines (FSM)**

  also known as Finite State Automata (FSA)
  system moves from one state to another depending on a number of defined conditions
  very similar to state chart - **move from state to state**
  context relevant (order matters)

  - characteristics



the new state it changes to is based on the input value as well as the current state (ballpoint pen)

despite the input being the same the results can be different since the current state needs to be taken into account as well

- a fixed set of possible state

  Finite states

- a set of input
- a set of possible output

  actually do not need to have outputs (automata)

- usage scenarios
  - recognize a pattern of incoming data

    spell checking (context aware manner)
    predictive text

- 2 kinds of diagrams representing FSMs
  - state transition diagram
    - initial state (only 1)
    - **accepting state (goal state)**
    - transaction
    - Example

      REGULAR EXPRESSION = ab*

      - start from initial state (0)
        - enter a

          system goes to accepting state (1)

          - enter b

            stay at accepting state 1

          - enter a

            goes to state 2
            no arrow goes from state 2 to 1

stick here forever

- enter b

  goes to state 2
  no arrow goes from state 2 to 1
  stick here forever

- un-accepting state (2)

  no matter what you input (a or b), the system will never change state

- state transition table

- **Variants of FSMs**

  provide more formal approach to describe the internal behavior of the system
  multiple start states
  the differences only in the way the output is generated

  - **Mealy machines**

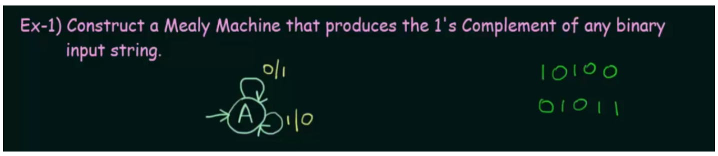
    the output is associated to the input
    the output is the function of present state as well as the input
    computationally equivalent to Moore machine
    a little bit more powerful and adaptive
    the output is in the transition from one state to another (flexible)

    - characteristics
      - Transitions are labelled *i/o*
        - *i* is a character in the input alphabet and
        - *o* is a character in the output alphabet.
      - no accept states

        it is not a language recognizer, it is an *output producer*

      - the output will be same length as it input

    - Example 1

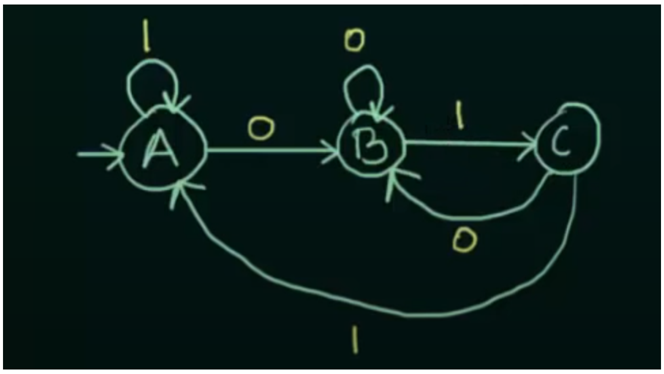
      

      Mealy machine takes the one's complement of its binary input. In other words, it flips each digit from a 0 to a 1 or from a 1 to a 0.

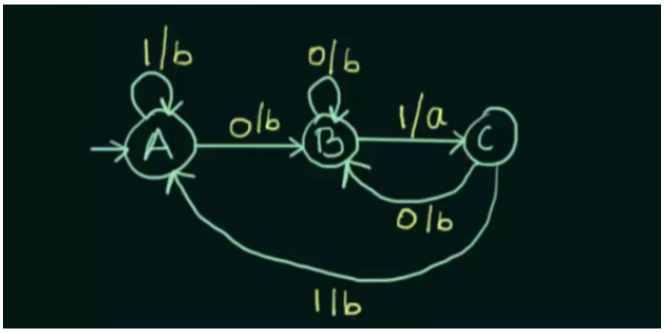
    - Example 2

      Construct a Mealy Machine that prints 'a' whenever the sequence '01' is encountered in any input binary string
      (and)

      - step 1: get DFA



- step 2: transfer DFA to Mealy machine



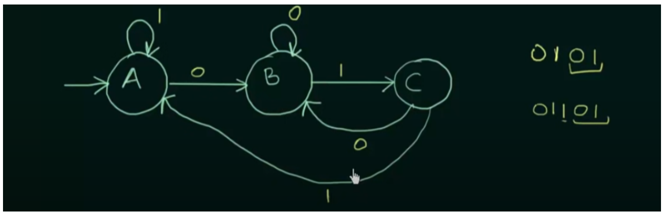
- **Moore machines**

  a finite state automaton with 2 extra attributes (input and output alphabet)
  the output is associated to the state
  the output is the function of present state
  may need more state

  - Example

    Construct a Moore Machine that prints 'a' whenever the sequence '01' is encountered
    in any input binary string

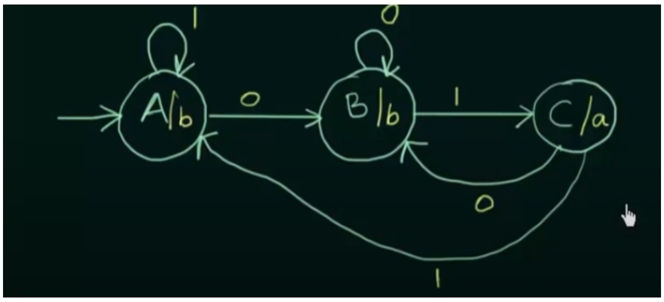
    - step 1: get DFA

      - start from state A
        - if enter 0 then go to state B
        - if enter 1 then stay in this state
      - (encounter 0) goes to state B
        - if enter 1 then go to state C
        - if enter 0 then stay in this state
      - (state B encounter 1) goes to state C
        - if enter 1 then system needs to wait for next 0 therefore goes to state A

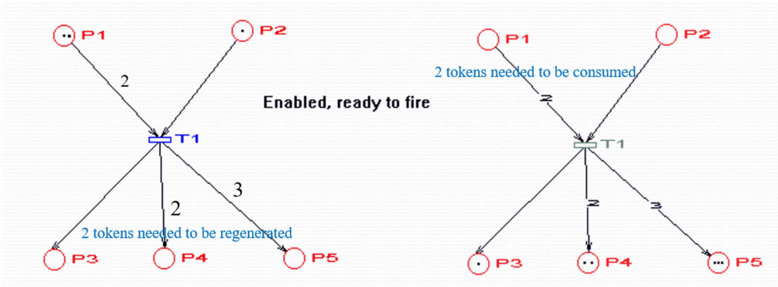
            - if enter 0 then system needs to wait for next 1 therefore goes to state B

        - step 2: transfer DFA to moore machine

          the output is associated to the state

- **Infinite State Machines**
    - **Petri Nets** (allow to be **non-deterministic**)

      can have infinite number of state (powerful)
      have the ability to move to **unpredictable state**
      provide more formal approach to describe the internal behavior of the system
      be used to model discrete distributed systems.

        - characteristics
            - components
                - places

                  Storage location
                  have infinite capacity by default

                - tokens

                  Token make something happen
                  tokens do not move
                  they disappear from the input when fire and **regenerate** on the output
                  the number of tokens in each of its input places is at least equal to the arc weight going from the place to the transition.
                  tokens are not conserved, the total number of tokens at input and output can change

                - transitions

                  Processes
                  square or solid bar

    transition can fire
    have no capacity, cannot store tokens

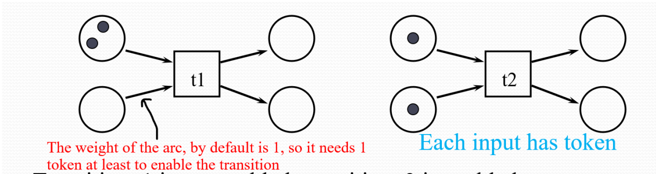
  - arcs (connections)

    arcs have capacity/weight (default is 1)
    can only connect places to transitions

  - inhibition

- **enable** condition

  - **每一个 arc 都有 weight (capacity)，如果 arc 源头的 place 里面的 token 数目大于等于 arc的 weight，那么transition is enabled**

    Transition t1 is not enabled, transition t2 is enabled.

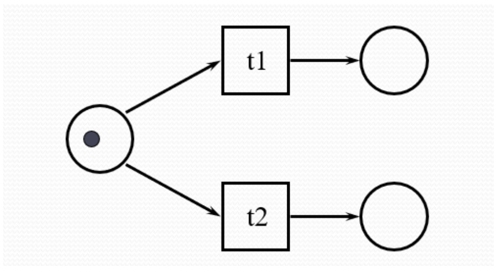
    A transition is **enabled** if each of the input places contains tokens.

- fire

  - enabled transition may fire, but we don't know when

  - firing is atomic

    - transitions cannot be fired at the same time

      either T1 or T2 will fire (represent non-deterministic)
      Even if there are two tokens, there is still a conflict.

  - firing: consuming tokens at the inputs and regenerating (producing) tokens at the output

    - special case

      - **without input (Permanent enable)**

        can ire at any time and produces tokens in the connected places:

      - ~~without output (must be enabled to fire)~~

        keep firing and consuming tokens until deadlocks

- high-level petri net

  - transition

- transition can have some constraint
- token - color
  - color
    - tokens represent objects, then color represents the **attributes**
- transition - time
  - time
    - t_{min} and t_{max}

      tell us the minimum and maximum time that a transition will take to fire once enabled.
- hierarchy
  - sub-net

    do not need to have all the details

- Entity-relation-attribute model (data base or class design)
  - Semantic data models (8)

    describe the logical structure of data which is imported to or exported by the systems.
    Widely used in database design
  - Data Dictionary
- Structural model
- **Object models**

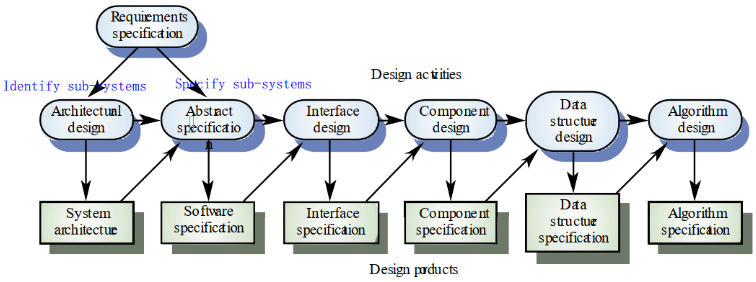
  describe the system in terms of object classes
  - **Unified Modeling Language (UML)**
    - use case diagrams
    - class diagrams
    - sequence diagrams
    - statechart diagrams
    - deployment diagrams

- design methodology

  requirements broken into small pieces (maybe break into classes)
  design: find a solution to both functional requirements and non-functional requirements
  - a good software
    - re-usable
    - understandable
    - flexible (changeable)
    - simple
    - portable (use on different system)
    - testable

- stages of design
    - problem understanding
    - Identify one or more solutions
    - Describe solution abstractions

        for example, use class diagram
        describe something graphically
        Any design may be modeled as a directed graph
    - Repeat process for each identified abstraction
- phases of design



- design methods
    - modular design
        - modularity : cheap and efficient and high quality
        - goal
            - what the modules are

                data and classes.etc
            - what modules do
            - interaction between modules
        - modular programming
            - break into **subroutines**
            - requirement
                - modules must be autonomous coherent
    - Procedural Abstraction

- **Implement**
    - convert structure into executable program
    - programming and Debugging
        - apply organisational standards
        - program testing

            discover faults and remove
            - testing stages
                - Unit testing
                - Module testing

related (dependent) subs are tested

- Sub-system testing (merges with system testing)

  Modules are integrated into sub-systems and tested. The focus here should be on **interface testing**

- System testing
- Acceptance testing

  Testing with customer data

- Good programming is **iterative**

  write part 1 code → test it → part 2 code → test → ....

- **validation**
- **evolution**