(T.A.O)

SYSTEM MODEL

At an early stage in the requirements elicitation and analysis process, the boundaries of the system are decided. This involves the system developer working strictly with other system stakeholders (e.g., managers, customers, and end users) to distinguish what the system is, what the system's environment is, and how they would relate together. These decisions should be made early in the process to limit system costs and the time needed for analysis.

In some cases, the boundary between a system and its environment is relatively clear. For example, where an automated system is replacing an existing manual or computerized system, the environment of the new system is usually the same as the existing system's environment.

In other cases, there is more flexibility, and you decide what constitutes the boundary between the system and its environment during the requirements engineering process.

System models basically show the relationships between the system components and the system and its environment. These might be object models, data-flow models and semantic data models.

System models are mostly used to reduce the ambiguity of the system to be developed and to communicate in a precise way the functions that the system must provide.

- Context models:

A context model (or context modelling) defines how context data are structured and maintained. The context is the surrounding element for the system, and a model provides the mathematical interface and a behavioural description of the surrounding environment.

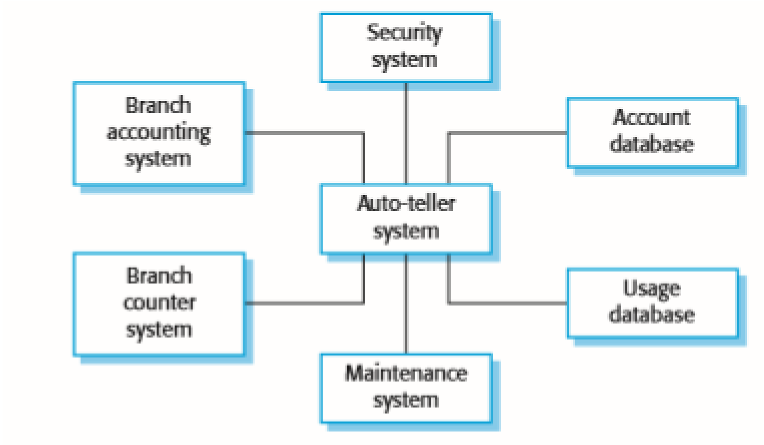
A system context diagram (SCD) in engineering is a diagram that defines the boundary between the system, or part of a system, and its environment, showing the entities that interact with it. This diagram is a high-level view of a system. It is similar to a block diagram. System context diagrams show a system, as a whole and its inputs and outputs from/to external factors.

Context diagrams can be developed with the use of two types of building blocks:

- Entities (Actors): labelled boxes, one in the centre representing the system, and around it multiple boxes for each external actor
- Relationships: labelled lines between the entities and system.

Architectural models are more of a context model, and they describe the environment of a system. However, they do not show the relationships between the other systems in the environment and the system that is being specified. External systems might produce data for or consume data from the system. They might share data with the system, or they might be connected directly, through a network, or not at all. They might be physically co-located or located in separate buildings. All of these relations might affect the requirements of the system being defined and must be taken into account.

Therefore, simple architectural models are normally supplemented by other models, such as process models, that show the process activities supported by the system.



The above diagram is an architectural model that illustrates the structure of the information system that includes a bank auto-teller network. High-level architectural models are usually expressed as simple block diagrams where each sub-system is represented by a named rectangle, and lines indicate associations between sub-systems. From the diagram above, we see that each ATM is connected to an account database, a local branch accounting system, a security system and a system to support machine maintenance. The system is also connected to a usage database that monitors how the network of ATMs is used and to a local branch counter system. This counter system provides services such as backup and printing. These, therefore, need not be included in the ATM system itself.

- Behavioural models

Behavioural models are used to describe the overall behaviour of the system. There are two types of behavioural models here:

- data-flow models, which model the data processing in the system, and
- state machine models, which model how the system reacts to events.

These models may be used separately or together, depending on the type of system that is being developed.

Most business systems are primarily driven by data. They are controlled by the data inputs to the system, with relatively little external event processing. A data-flow model may be all that is needed to represent the behaviour of these systems.

By contrast, real-time systems are often event-driven with minimal data processing. A state machine model is the most effective way to represent their behaviour.

Other classes of systems may be both data and event-driven. In these cases, you may develop both types of models.

2.1. Data-flow models

(T.A.O)

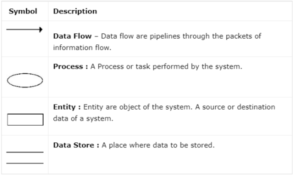
Data-flow models are an intuitive way of showing how data is processed by a system. At the analysis level, they should be used to model the way in which data is processed in the existing system.

Data-flow models are used to show how data flows through a sequence of processing steps. These processing steps or transformations represent software processes or functions when data-flow diagrams are used to document a software design.

Data-flow models are valuable because tracking and documenting how the data associated with a particular process moves through the system helps analysts understand what is going on. Data-flow diagrams have the advantage that, unlike some other modelling notations, they are simple and intuitive. It is usually possible to explain them to potential system users, who can then participate in validating the analysis.

Data-flow models show a functional perspective where each transformation represents a single function or process. They are particularly useful during the analysis of requirements as they can be used to show end-to-end processing in a system. They show the entire sequence of actions that take place from an input being processed to the corresponding output which is the system's response.

N: B- Data Flow Diagram was designed to represent the exchange of information. Connectors in a Data Flow diagram are for representing data, not process flow, steps, or anything else. When we label a data flow that ends at a data store "a request", this means we are passing a request as data into a data store. Although this may be the case at the implementation level as some of the DBMS do support the use of functions, which intake some values as parameters and return a result, in Data Flow Diagram, we tend to treat the data store as a sole data holder that does not possess any processing capability.



Symbol	Description
→	**Data Flow** – Data flow are pipelines through the packets of information flow.
⬭	**Process :** A Process or task performed by the system.
▭	**Entity :** Entity are object of the system. A source or destination data of a system.
=	**Data Store :** A place where data to be stored.

Here's a simplified example of a Data Flow diagram (DFD) for an online shopping system:

1. Entities:

Customer

2. Processes:

- Order Processing
- Inventory Management
- Payment Processing

3. Data Stores:

- Product Database

- Order Database

Inventory Database

4. Data Flows:

- Customer submits order (arrow from Customer to Order Processing)

- Order details are sent to Inventory Management (arrow from Order Processing to Inventory Management)

- Inventory data is updated (arrow from Inventory Management to Inventory Database)

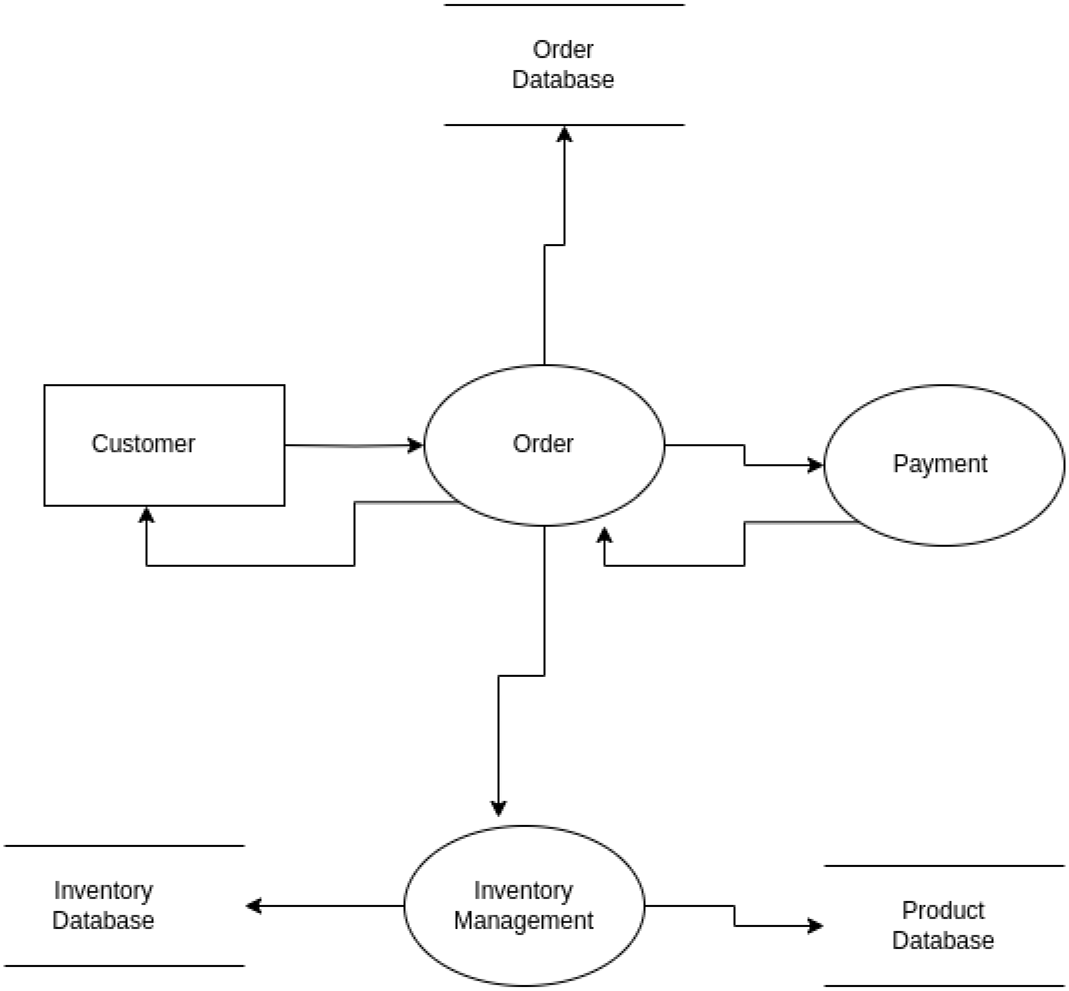
- Payment information is sent to Payment Processing (arrow from Order Processing to Payment Processing)

- Payment status is updated (arrow from Payment Processing to Order Processing)

- Order details are stored in the Order Database (arrow from Order Processing to Order Database)

- Product information is retrieved from the Product Database (arrow from Inventory Management to Product Database)

- Order status is sent to the Customer (arrow from Order Processing to Customer)



This DFD illustrates how data (orders, product details, payments) flows between various components (customer, databases, payment gateway) and processes (order processing, inventory management, payment processing) within an online shopping system. It simplifies the system's functionality for better understanding and analysis.

2.2 State Machine Model

A State Machine Model, is a computational model used to represent the behavior of a system, process, or object that can exist in a limited number of discrete states.

A computational model is an abstract representation or simulation of a real-world system, process, or phenomenon that is designed to be analyzed or studied using a computer.

Here's an overview of the key components and concepts of a state machine model:

States: States represent the different conditions or modes that a system or object can be in. These states are discrete and well-defined. For example, a traffic light can be in states like "Green," "Yellow," or "Red."

Transitions: Transitions are the events or conditions that cause a system or object to change from one state to another. Transitions are triggered by specific events, input signals, or conditions. For example, a traffic light transitions from "Green" to "Yellow" when a timer expires.

Events: Events are external stimuli or triggers that initiate state transitions. They can be physical inputs (e.g., button presses, sensor readings), messages, or other signals from the environment. Events cause the state machine to react and change states.

Actions: Actions or behaviours are associated with states or transitions. They specify what happens when a system or object enters or exits a particular state or when a transition occurs. Actions can include performing specific tasks, sending messages, or changing variables.

Initial State: Every state machine has an initial state, which represents the starting point when the system or object is initialised or triggered. It's the state where the system begins its operation.

Final State: A final state represents the end of the state machine's operation or the termination of a process. When the system reaches a final state, it indicates that a specific task or process has been completed.

Hierarchical States: State machines can have hierarchical structures, where states can contain sub-states, forming a nested hierarchy. This helps in organising and modelling complex systems with multiple levels of behaviour.

Parallel States: State machines can also handle parallel behaviour, where multiple states or sub-state machines can operate concurrently. This is useful for modelling systems with concurrent processes.

Guard Conditions: Guard conditions are logical expressions associated with transitions. They determine whether a transition can occur based on specific criteria or conditions being met.

History States: History states remember the previous state or sub-state within a hierarchical structure. This is useful for resuming behaviour from a previous point when re-entering a state.

State machine models are particularly valuable for designing systems with well-defined and predictable behaviour. They are commonly used in software engineering for designing control logic, user interfaces, and protocol handling. In hardware design, state machines are used for controlling digital circuits and sequential logic. They provide a structured way to represent and analyze complex system behaviour and are a fundamental concept in computer science and engineering.

A state machine model describes how a system responds to internal or external events. The state machine model shows system states and events that cause transitions from one state to another. It does not show the flow of data within the system. This type of model is often used for modelling real-time systems because these systems are often driven by stimuli from the system's environment.

(T.A.O)

Petri net

A Petri net is a 6-tuple (P, T, A, W, M0, I), where:

P is a finite set of places.

T is a finite set of transitions.

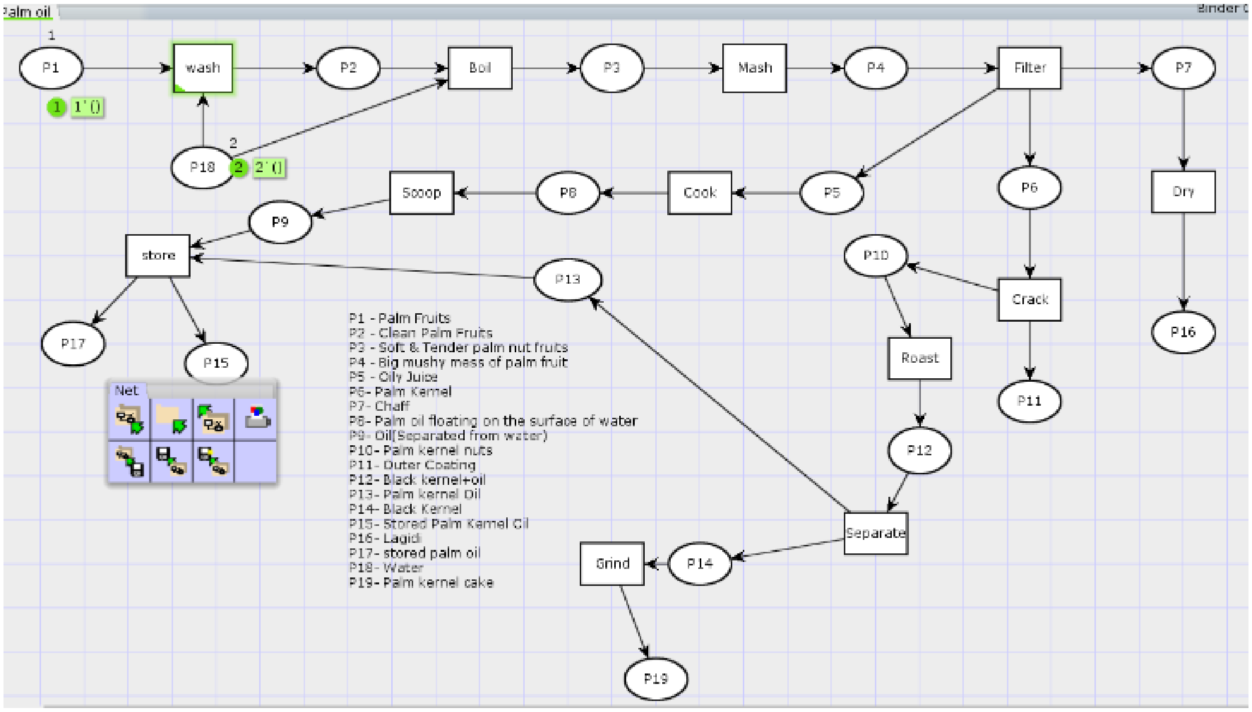
A is a finite set of arcs, where A ⊆ (P × T) ∪ (T × P) is a set of directed arcs connecting places to transitions or transitions to places.

W: A ⟶ {1, 2, 3, ...} is a weight function that assigns a weight to each arc, specifying how many tokens are needed to traverse it.

M0: P ⟶ {0, 1, 2, ...} is an initial marking that assigns the number of tokens to each place at the beginning of the Petri net's execution.

I ⊆ P is a set of input places, which defines a subset of places where tokens are consumed to enable the firing of transitions.

The Petri net represents a system where tokens move between places through transitions. The transitions can fire based on the availability of tokens in input places and the weights of arcs.



P1 - Palm Fruits
P2 - Clean Palm Fruits
P3 - Soft & Tender palm nut fruits
P4 - Big mushy mass of palm fruit
P5 - Oily Juice
P6- Palm Kernel
P7- Chaff
P8- Palm oil floating on the surface of water
P9- Oil(Separated from water)
P10- Palm kernel nuts
P11- Outer Coating
P12- Black kernel+oil
P13- Palm kernel Oil
P14- Black Kernel
P15- Stored Palm Kernel Oil
P16- Lagidi
P17- stored palm oil
P18- Water
P19- Palm kernel cake

The above diagram is a Petri net diagram that shows the model of the processes involved in domestic production of palm oil, palm kernel oil, and their associated by-products using coloured Petri Nets (CPN tools).

Petri net is a modelling tool used to

The Ps or places, represent the states

The transitions are in a rectangle shape to show the event that changes the states

Finite State Machine (FSM)

A finite state machine is a 6-tuple ($Q, \Sigma, \delta, \delta I, \delta F$), where:

Q is a finite set of states.

Σ is a finite alphabet, which represents the set of input symbols.

$\delta: Q \times \Sigma \rightarrow Q$ is a transition function that maps a current state and an input symbol to a new state.

$\delta I \subseteq Q$ is a set of initial states.

$\delta F \subseteq Q$ is a set of final (accepting) states.

$\Lambda: Q \times \Sigma \rightarrow \Lambda$ is an output function (optional) that maps a state and an input symbol to an output symbol.

The FSM models a system that can be in one of a finite number of states and transitions between states in response to input symbols. If an output function is included, it can also produce output symbols as the machine transitions. An FSM is used to describe the behaviour of systems with discrete and sequential logic.

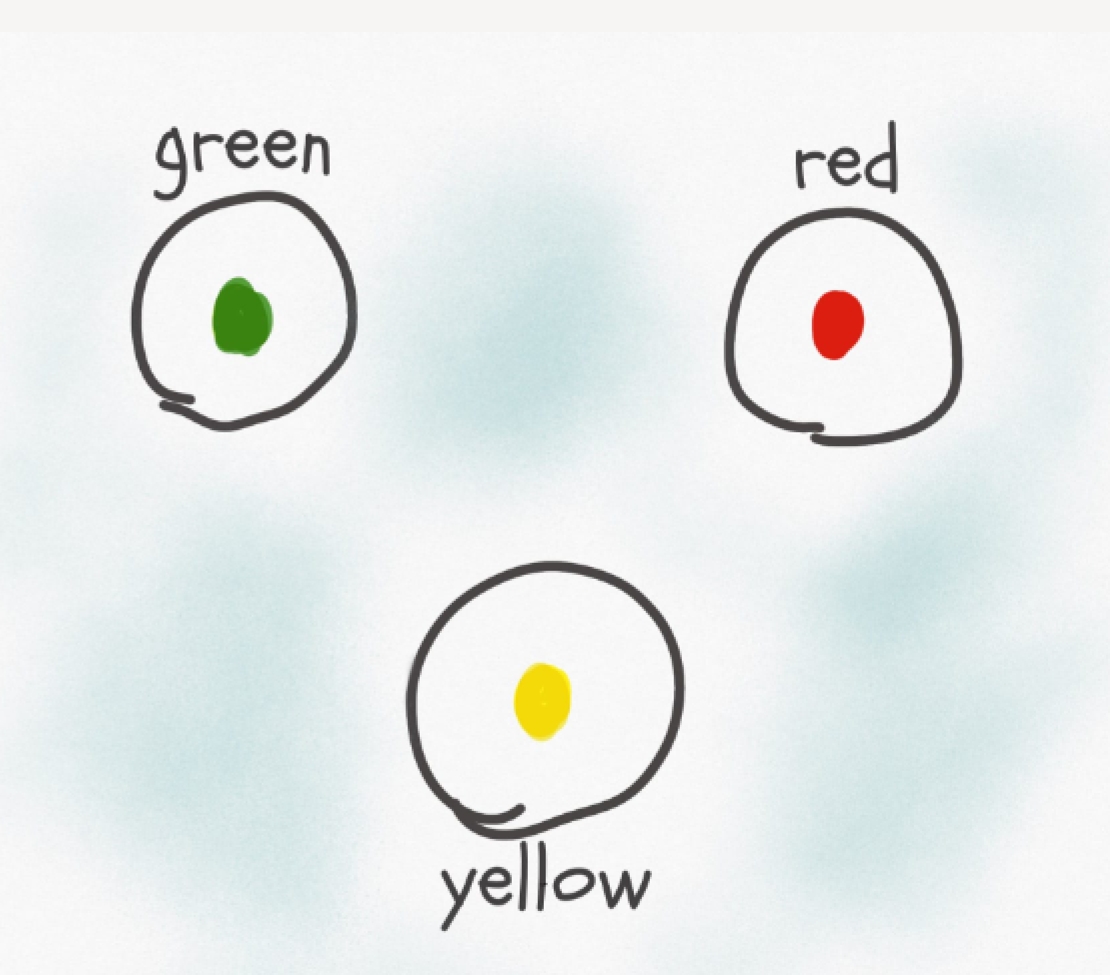
This is the most basic and widely used type of state machine. It consists of a finite set of states, transitions between these states triggered by events, and actions associated with transitions. FSMs are used to model sequential logic in hardware and control logic in software.

Example

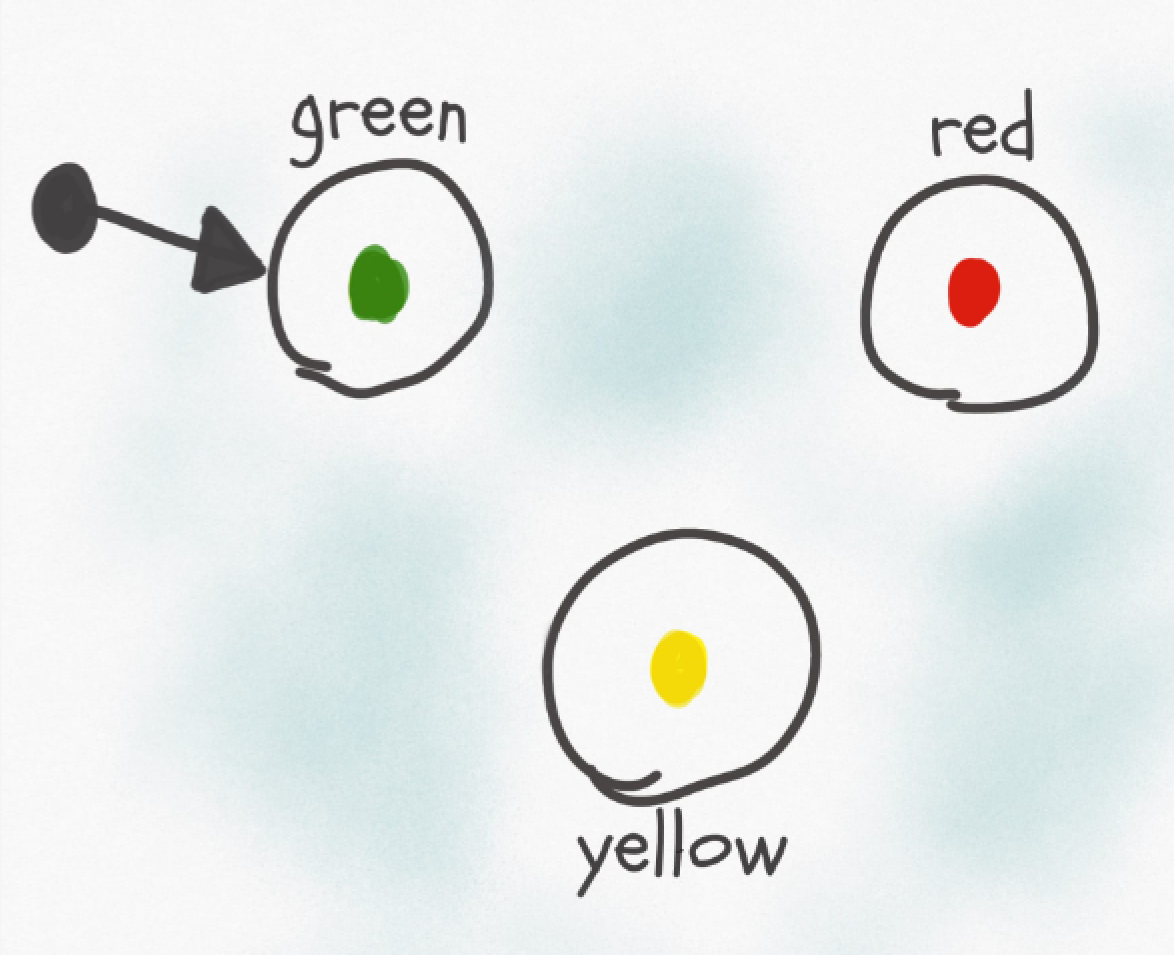
A traffic light state machine is said to be finite because we have a finite number of states. 3 states

- has the green light on, and the other 2 lights off
- has the red light on, and the other 2 lights off
- as the yellow light on, and the other 2 lights off

(T.A.O)



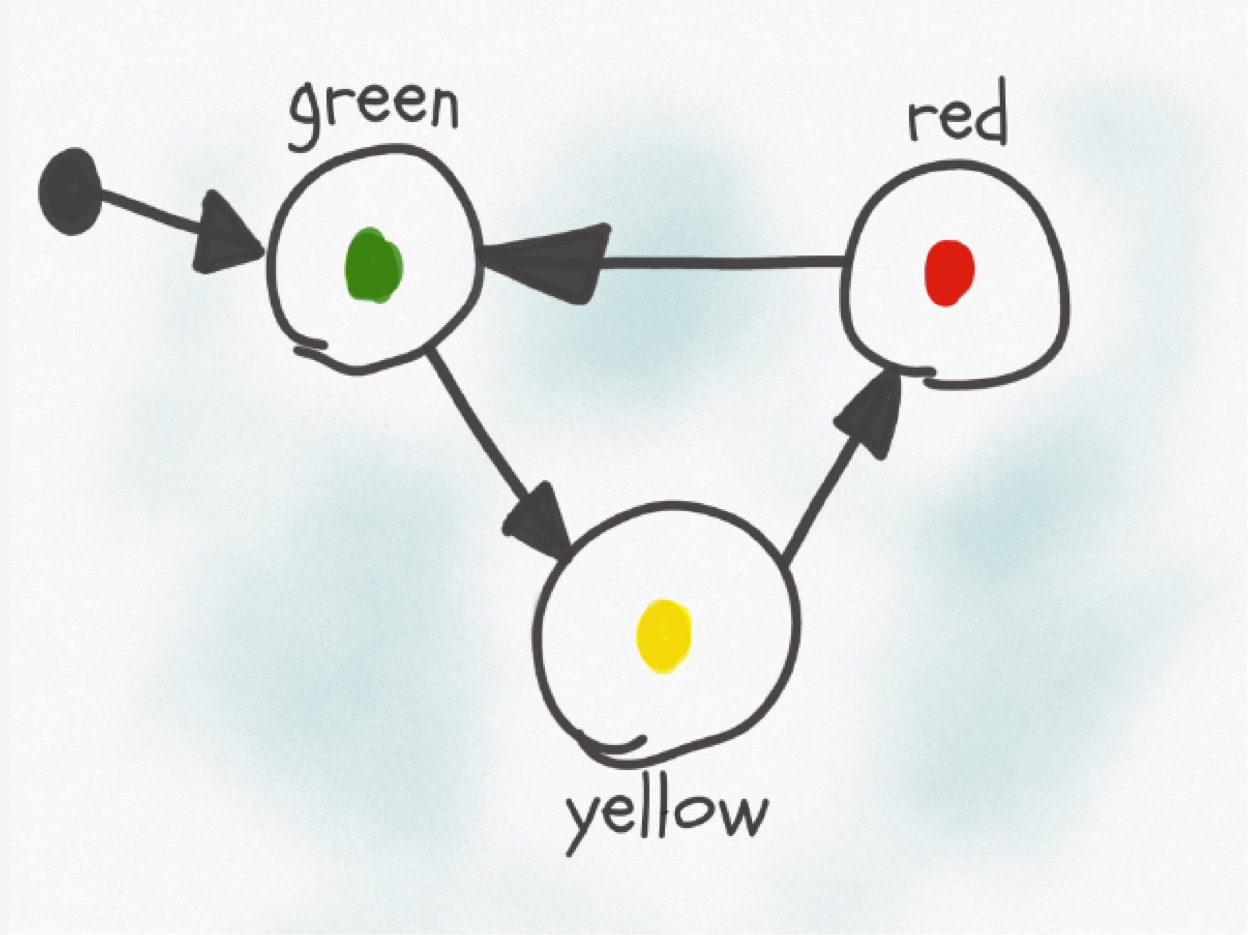
We have an initial state. Let’s say our traffic lights start, when we reset them, at the green state.



We have a timer that after 50 seconds of green state, moves the semaphore into the yellow state. We have 8 seconds of yellow state, then we move to the red state, and we sit there for 32 seconds, because that road is secondary and it deserves less time of green. After this, the semaphore gets back to the green state and the cycle continues indefinitely, until the electricity halts and the semaphore resets when it gets the power again.

In total, we have a 90 seconds cycle.

This is how we can model it:



We can also model it with a state transition table, which shows the state the machine is currently in, what is the input it receives, what is the next state, and the output:

State	Input	Next state	Output
Green	Timer 50s	Yellow	Green light on, others off
Yellow	Timer 8s	Red	Yellow light on, others off
Red	Timer 32s	Green	Red light on, others off

Assignment

For DFDs, discuss how data flows in an online shopping system.

For State Transition Diagrams, a vending machine.

(T.A.O)

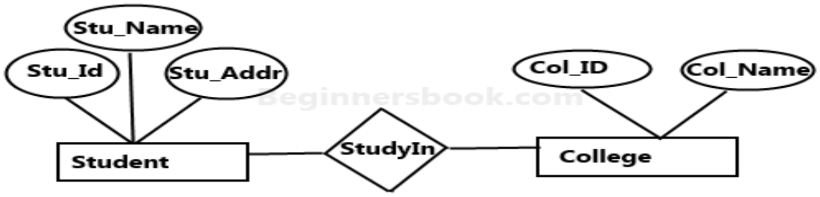
Entity Relationship Diagram

Most large software systems make use of a large database of information. In some cases, this database is independent of the software system. In others, it is created for the system being developed. An important part of systems modelling is defining the logical form of the data processed by the system. These are sometimes called semantic data models.

The most widely used data modelling technique is Entity-Relation-Attribute modelling (ERA modelling), which shows the data entities, their associated attributes and the relations between these entities.

Entity-relationship models have been widely used in database design. The relational database schemas derived from these models are naturally in third normal form, which is a desirable characteristic. Because of the explicit typing and the recognition of sub- and super-types, it is also straightforward to implement these models using object-oriented databases. The UML does not include a specific notation for this database modelling, as it assumes an object-oriented development process and models data using objects and their relationships. However, you can use the UML to represent a semantic data model. You can think of entities in an ERA model as simplified object classes (they have no operations), attributes as class attributes and named associations between the classes as relations.

A simple ER Diagram



The components and features of an ER diagram

Entity

A definable thing—such as a person, object, concept or event—that can have data stored about it. Think of entities as nouns. Examples: a customer, student, book, car or product. Typically shown as a rectangle.

Attribute

A property or characteristic of an entity. Often shown as an oval, circle or inside the rectangle that describes the entity.

Relationship

How entities act upon each other or are associated with each other. Think of relationships as verbs. For example, the named student might register for a course. The two entities would be the student and the course, and the relationship depicted is the act of enrolling, connecting the two entities in that way. Relationships are typically shown as diamonds or labels directly on the connecting lines.

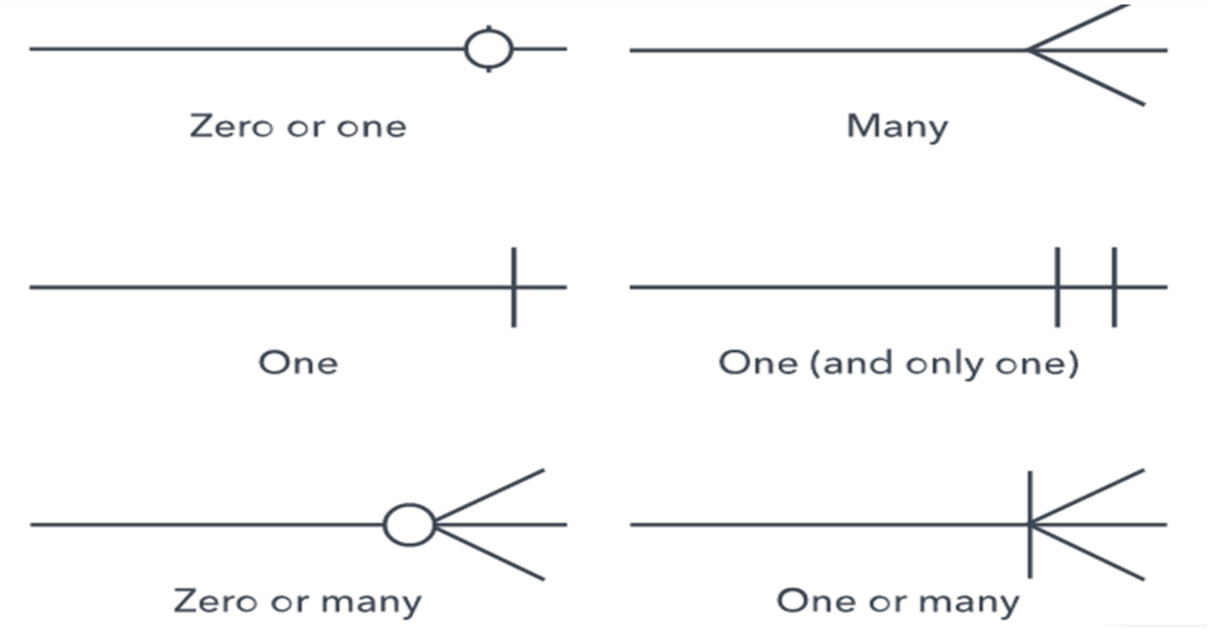
Cardinality

Defines the numerical attributes of the relationship between two entities or entity sets. The three main cardinal relationships are one-to-one, one-to-many, and many-many.

A **one-to-one example** would be one student associated with one mailing address.

A **one-to-many example (or many-to-one, depending on the relationship direction):** One student registers for multiple courses, but all those courses have a single line back to that one student.

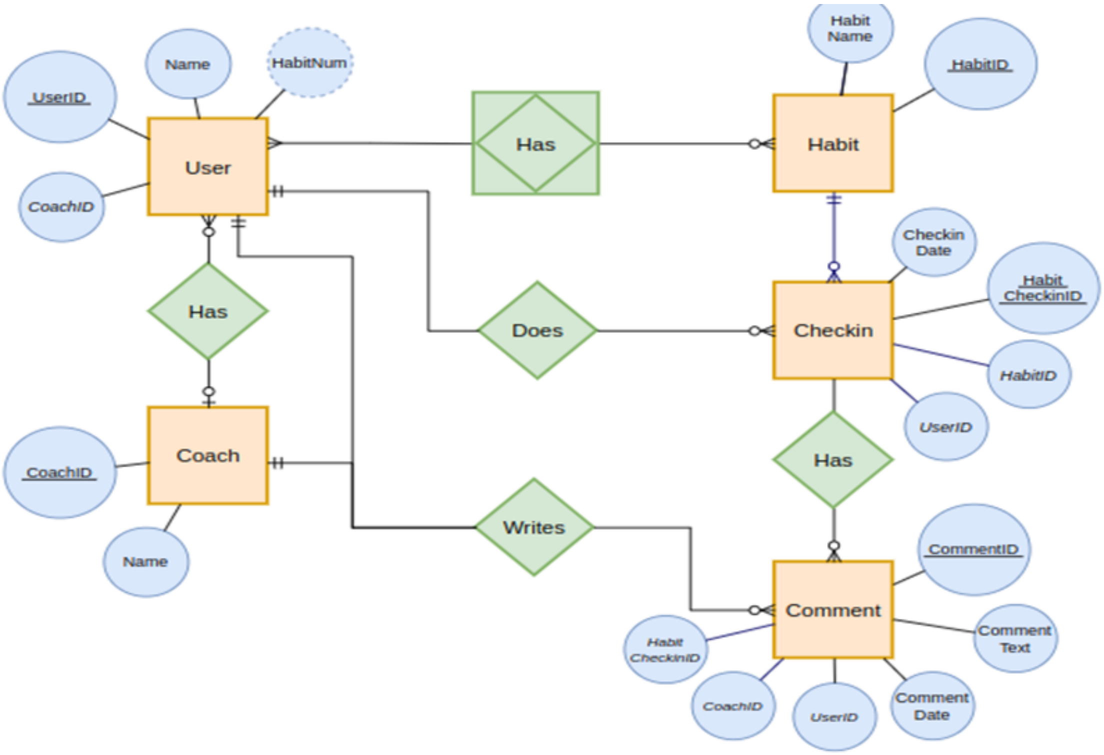
Many-to-many example: Students as a group are associated with multiple faculty members, and faculty members in turn are associated with multiple students.



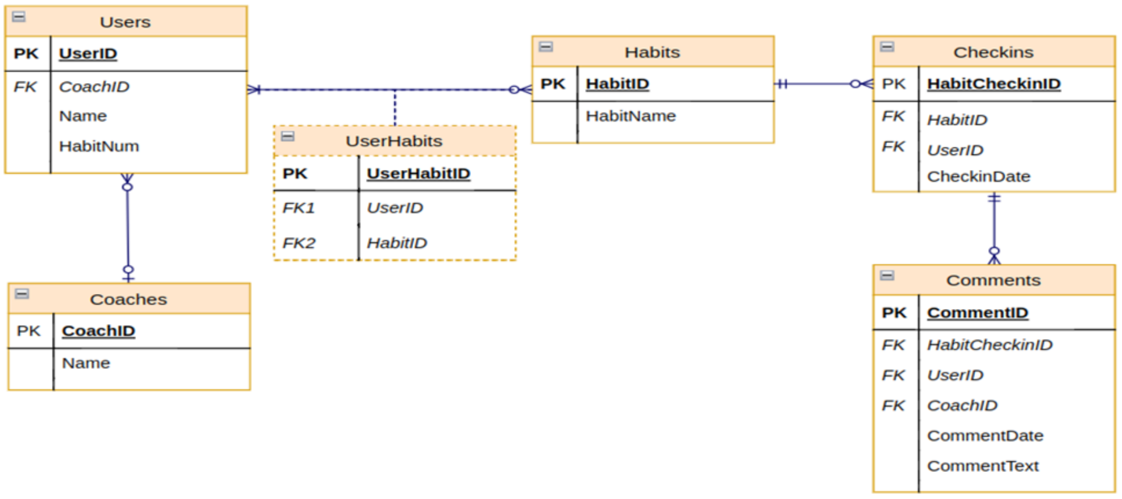
ERD symbols and notations

There are several notation systems, which are similar but vary in a few specifics, but the most common are the Chen and Crow’s foot notations.

Example of Chen notation,



Example of Crow notation,



Users
PK UserID
FK CoachID
Name
HabitNum

Habits
PK HabitID
HabitName

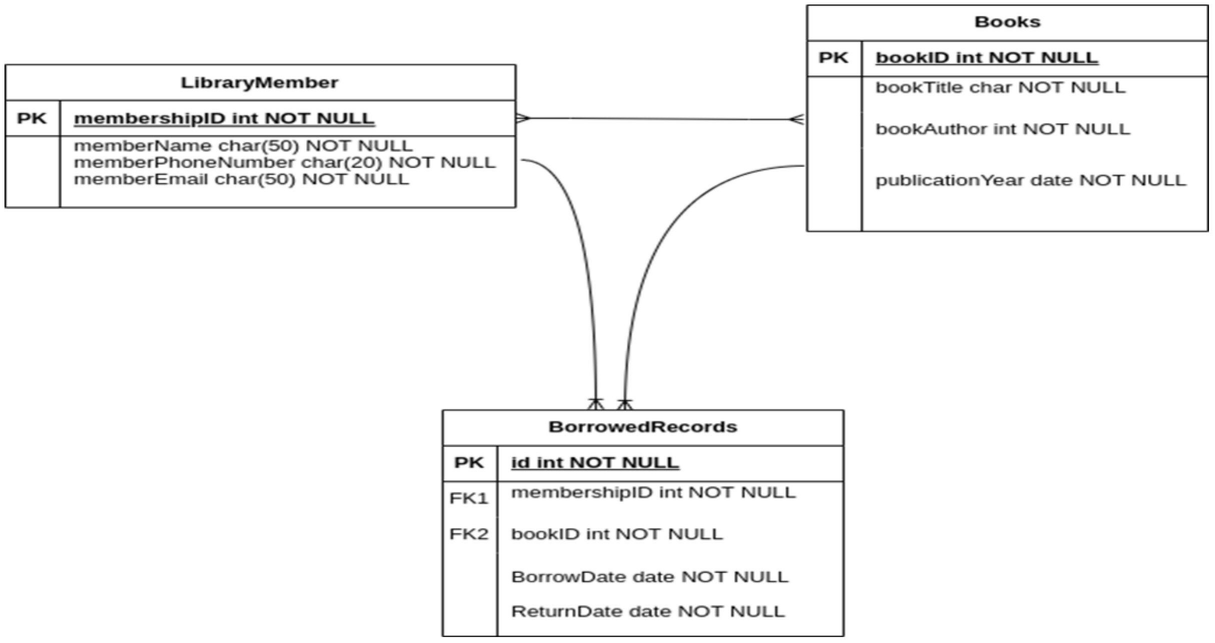
Checkins
PK HabitCheckinID
FK HabitID
FK UserID
CheckinDate

UserHabits
PK UserHabitID
FK1 UserID
FK2 HabitID

Coaches
PK CoachID
Name

Comments
PK CommentID
FK HabitCheckinID
FK UserID
FK CoachID
CommentDate
CommentText

Representing our library example using the Crow foot notation



Books
PK bookID int NOT NULL
bookTitle char NOT NULL
bookAuthor int NOT NULL
publicationYear date NOT NULL

LibraryMember
PK membershipID int NOT NULL
memberName char(50) NOT NULL
memberPhoneNumber char(20) NOT NULL
memberEmail char(50) NOT NULL

BorrowedRecords
PK id int NOT NULL
FK1 membershipID int NOT NULL
FK2 bookID int NOT NULL
BorrowDate date NOT NULL
ReturnDate date NOT NULL

- **Object models**

An object-oriented approach to the whole software development process is now commonly used, particularly for interactive system development.

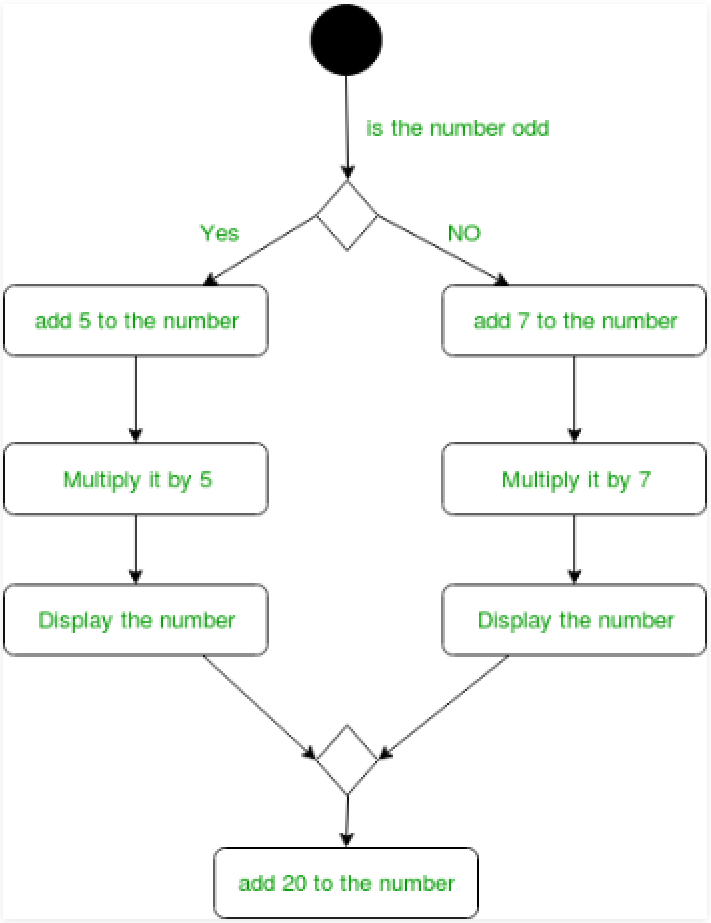
An interaction diagram is used to show the interactive behaviour of a system. Since visualizing the interactions in a system can be a cumbersome task, we use different types of interaction diagrams to capture various features and aspects of interaction in a system.

UML Activity Diagram

Activity diagrams are graphical representations of workflows of stepwise activities and actions with support for choice, iteration, and concurrency. In the Unified Modeling Language, activity diagrams are intended to model both computational and organizational processes (i.e., workflows), as well as the data flows intersecting with the related activities. Although activity diagrams primarily show the overall flow of control, they can also include elements showing the flow of data between activities through one or more data stores.

Activity diagrams are constructed from a limited number of shapes, connected with arrows. The most important shape types are:

- rectangle with rounded corners represents an activity which represents execution of an action on objects or by objects.;
- diamonds represent decisions.
- A black circle represents the start (initial node) of the workflow;
- An encircled black circle represents the end (final node).

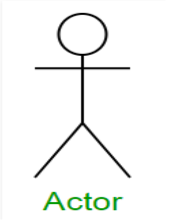


Sequence diagram

A sequence diagram shows the sequence of events. It shows how object in a system or classes interact with each other.

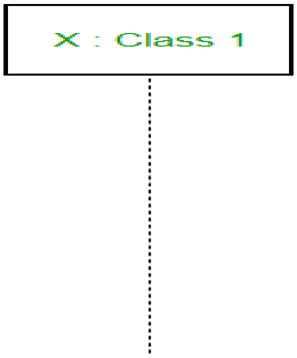
Sequence Diagram Notation

1. Actors – An actor in a UML diagram represents a type of role where it interacts with the system and its objects. It is important to note here that an actor is always outside the scope of the system we aim to model using the UML diagram.



We use actors to depict various roles including human users and other external subjects. We represent an actor in a UML diagram using a stick person notation. We can have multiple actors in a sequence diagram.

2. Lifelines – A lifeline is a named element which depicts an individual participant in a sequence diagram. So basically each instance in a sequence diagram is represented by a lifeline. Lifeline elements are located at the top in a sequence diagram. The standard in UML for naming a lifeline follows the following format – Instance Name : Class Name.



Assignment

1. Find out other sequence diagram notations.
2. Use a sequence diagram to show a graphical representation of any simple computable solvable problem you can think of.

(Note: I expect that the problem represented shouldn’t be the same for two students)

Class Diagram

The UML Class diagram is a graphical notation used to construct and visualize object oriented systems. A class diagram in the Unified Modeling Language (UML) is a type of static structure diagram that describes the structure of a system by showing the system's:

- classes,
- their attributes,
- operations (or methods),
- and the relationships among objects.

What is a Class?

A Class is a blueprint for an object. Objects and classes go hand in hand. We can't talk about one without talking about the other. And the entire point of Object-Oriented Design is not about objects, it's about classes, because we use classes to create objects. So a class describes what an object will be, but it isn't the object itself.

Classes describe the type of objects, while objects are usable instances of classes. Each Object was built from the same set of blueprints and therefore contains the same components (properties and methods).

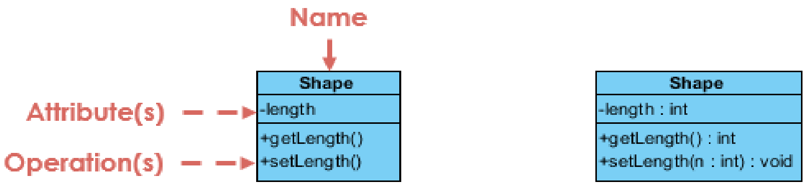
(T.A.O)

The standard meaning is that an object is an instance of a class and object - Objects have states and behaviors.

CLASS NOTATION

A class notation consists of three parts:

1. Class Name
    - The name of the class appears in the first partition.
2. Class Attributes
    - Attributes are shown in the second partition.
    - The attribute type is shown after the colon.
    - Attributes map onto member variables (data members) in code.
    - Attributes are a significant piece of data containing values that describe each instance of that class
    - Also called fields, variables or properties
3. Class Operations (Methods or functions)
    - Operations are shown in the third partition. They are services the class provides i.e. specifies any behavioral feature of the class.
    - The return type of a method is shown after the colon at the end of the method signature.
    - The return type of method parameters is shown after the colon following the parameter name.
    - Operations map onto class methods in code

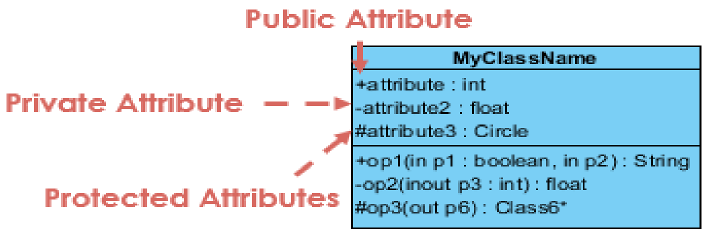


Class without signature

Class with signature

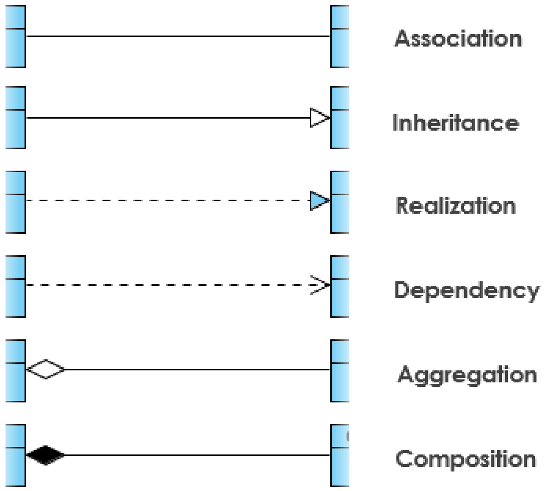
Class Visibility

The +, - and # symbols before an attribute and operation name in a class denote the visibility of the attribute and operation.



- + denotes public attributes or operations i.e. the exact opposite of private. It can be accessed by any other class or subclass.
- - denotes private attributes or operations i.e. they cannot be accessed by any other class or subclass.
- # denotes protected attributes or operations i.e. they can only be accessed by the same class or subclasses
- There's another type of class visibility denoted as ~. It represents package/default. This means it can be used by any other class as long as they are in the same package. This rarely ever used.

Class Relationship

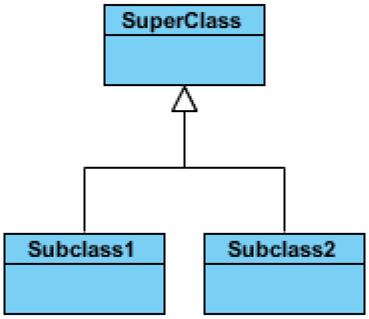


The figure above represents the arrows depicting each of the relationship types between class.

(T.A.O)

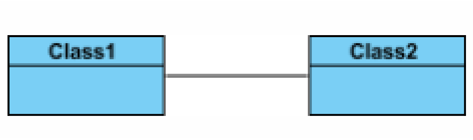
Relationship types in details

1. Inheritance (or Generalization):



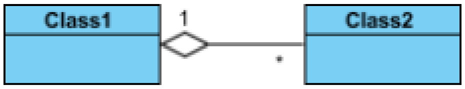
- Represents an "is-a" relationship.
- An abstract class name is shown in italics or << >>
- SubClass1 and SubClass2 are specializations of SuperClass i.e. they are derived from SuperClass.
- SuperClass is the parent class while Subclass1 and Subclass2 are both child classes.

2. Simple Association



Associations are relationships between classes in a UML Class Diagram. They are represented by a solid line between classes. Associations are typically named using a verb or verb phrase which reflects the real-world problem domain.

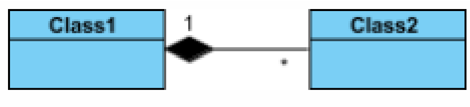
3. Aggregation



Aggregation is a special type of association. It represents a "part of" relationship i.e. a relationship where a part exists outside the whole.

- Class2 is part of Class1.
- Objects of Class1 and Class2 have separate lifetimes.

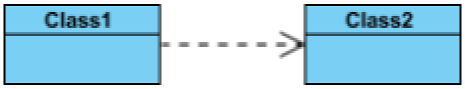
4. Composition



Composition is a special type of aggregation where parts are destroyed when the whole is destroyed.

- Objects of Class2 live and die with Class1.
- Class2 cannot stand by itself except if Class1 exists.

5. Dependency



- Exists between two classes if the changes to the definition of one may cause changes to the other (but not the other way around).
- Class1 depends on Class2
- A dashed line with an open arrow

Other relationship types here are multiplicity and realization.

Assignment

Read up on these other relationship type and briefly explain what they are.

References

https://beginnersbook.com/2015/04/e-r-model-in-dbms/

https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/

https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-class-diagram/

Abbott, R. (1983). Program design by informal English descriptions. Comm. ACM, 26(11), 882–94. (Ch. 14)

Abdel-Ghaly, A. A., Chan, P. Y., et al. (1986). Evaluation of competing software reliability predictions. IEEE Trans. on Software Engineering, SE-12(9), 950–67. (Ch. 24)

Ackroyd, S., Harper, R., et al. (1992). Information Technology and Practical Police Work. Milton Keynes: Open University Press. (Ch. 2)

Adams, E. N. (1984). Optimizing preventative service of software products. IBM J. Res. & Dev., 28(1), 2–14. (Ch. 3)

Ahern, D. M., Clouse, A., et al. (2001). CMMI Distilled. Reading, MA: Addison-Wesley. (Chs. 28, 29)

Albrecht, A. J. (1979). Measuring application development productivity. Proc. SHARE/ GUIDE IBM Application Development Symposium. (Ch. 26)

Albrecht, A. J. and Gaffney, J. E. (1983). Software function, lines of code and development effort prediction: a software science validation. IEEE Trans. on Software Engineering, SE-9(6), 639–47. (Ch. 26)

Adeniyi, O. R., Ogunsola, G. O., & Oluwusi, D. (2014). Methods of Palm Oil Processing in Ogun state, Nigeria: A Resource Use Efficiency Assessment. American International Journal of Contemporary Research, 4 (8): 173, 179.

Bagert, D. J. (2002). Texas licensing of software engineers: all's quiet for now. Comm. ACM, 45(11), 92–4. (Ch. 24)

Baker, F. T. (1972). Chief programmer team management of production programming. IBM Systems J., 11(1), 56–73. (Ch. 25)

Baker, T. (2002). Lessons learned integrating COTS into systems. Proc. ICCBSS 2002 (1st Int. Conf on COTS-based Software Systems), Orlando, FL: Springer-Verlag. (Ch. 18)

Balk, L. D. and Kedia, A. (2000). PPT: a COTS integration case study. Proc. Int. Conf. on Software Engineering, Limerick, Ireland: ACM Press. (Ch. 18)

Bamford, R. and Deibler, W. J., eds. (2003). ISO 9001: 2000 for Software and Systems Providers: An Engineering Approach. CRC Press. (Ch. 27)

Banker, R. D., Datar, S. M., et al. (1993). Software complexity and maintenance costs. Comm. ACM, 36(11), 81–94. (Chs. 21, 26)

Banker, R., Kauffman, R., et al. (1994). An empirical test of object-based output measurement metrics in a computer-aided software engineering (CASE) environment. J. of Management Info. Sys., 8(3), 127–50. (Ch. 26)

Bansler, J. P. and Bødker, K. (1993). A reappraisal of structured analysis: design in an organizational context. ACM Trans. on Information Systems, 11(2), 165–93. (Ch. 4)

Barker, R. (1989). CASE* Method: Entity Relationship Modelling. Wokingham: AddisonWesley. (Ch. 8)

Barnard, J. and Price, A. (1994). Managing code inspection information. IEEE Software, 11(2), 59–69. (Chs. 22, 27)

Basili, V. and Green, S. (1993). Software process improvement at the SEL. IEEE Software, 11(4), 58–66. (Ch. 28)

Ohimain, E. I., Izah, S. C., & Obieze, F. A. (2013). Material-mass balance of smallholder oil palm processing in the Niger Delta, Nigeria. Advance Journal of Food Science and Technology, 5(3), 289-294.

Gold, I. L., Ikuenobe, C. E., Asemota, O., & Okiy, D. A. (2012). Palm and Palm Kernel Oil Production and Processing in Nigeria. In Palm Oil (pp. 275-298).

Petri, C. A., & Reisig, W. (2008). Petri net. Scholarpedia, 3(4), 6477.

Murata T.,(1989). “Petri Nets:Properties, Analysis, And Applications”, Proceedings of IEEE, Vol 77, No 4, pp. 541-580

Zurawski, R., & Zhou, M. (1994). Petri nets and industrial applications: A tutorial. IEEE Transactions on industrial electronics, 41(6), 567-583

Taiwo, K. A., Owolarafe, O. K., Sanni, L. A., Jeje, J. O., Adeloye, K., & Ajibola, O. O. (2000). Technological assessment of palm oil production in Osun and Ondo states of Nigeria. Technovation, 20(4),