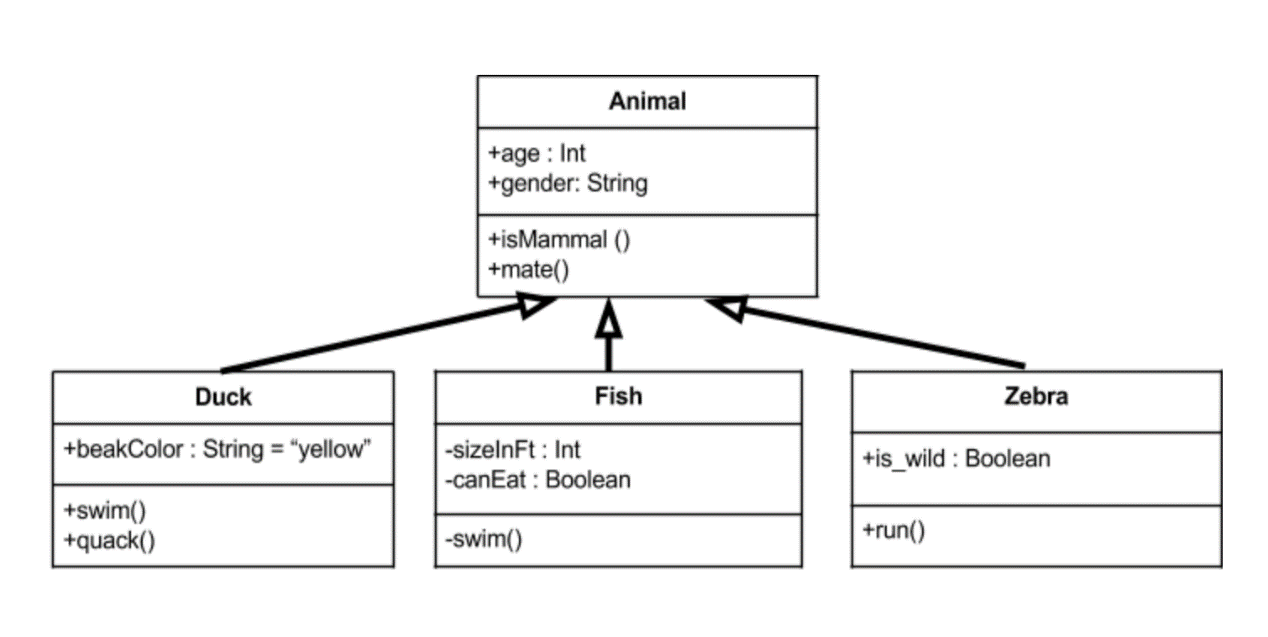
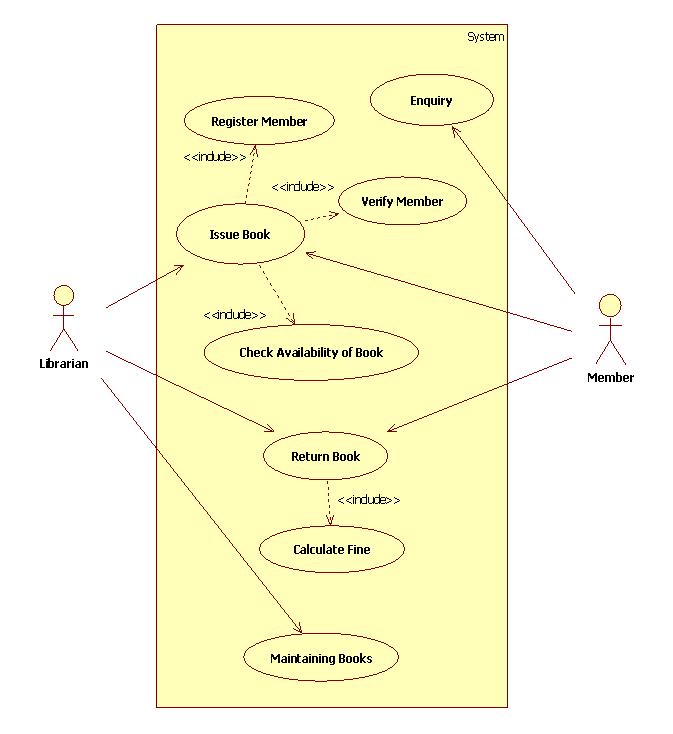
**UML (Unified Modelling Language)**

* Can be used as a notation for Object model in analysis and design.
* UML comprises of several diagrams as:
  + Class diagram (static)
  + Use case diagram (static)
  + Sequence diagram (dynamic)
  + State-chart diagram (dynamic)
  + Activity diagram (dynamic)
  + Collaboration diagram (static)
  + Implementation diagram
    - Component diagram
    - Deployment diagram
* **Class diagram**
  + Shows classes and the associations between those
  + Classes are defined by a set of attributes and operations
  + Cardinality of an association specifies how many instances are associated with a given number of source instances
  + There are four relationship types that can be shown in a class diagram
    - Association (plain line)
    - Aggregation (line with a diamond end from part to whole)
    - Dependency (dotted line with an arrow from dependent to required part)
    - Generalization (line with triangle arrowhead from subclass to superclass)



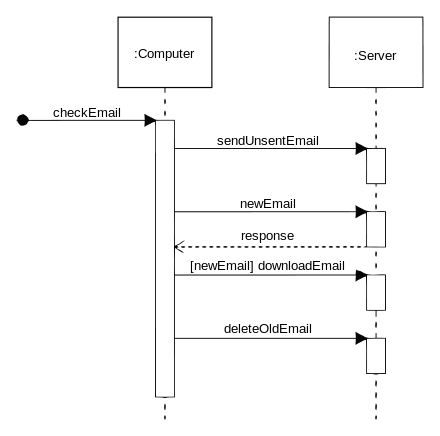
* + In composition, parts cannot exist without the whole. Represented with a solid diamond
  + Stereotype is a marker/label that is used to differentiate between objects existing at the same level. It labels a relationship between classes or objects (e.g. A --- <<uses>> --- B)
* **Use case diagram**

* + Shows relationships among actors and use cases. An actor stays outside of the system and interacts with the use case.
  + A use case is a unit of functionality of the system
  + Three types of relationships exist in a use case diagram
    - Communicates (between actor and use case)
    - Extends (between use cases, shown using generalization arrow)
    - Uses (between use cases, shown using generalization arrow)



* **Sequence diagram**

* + Shows simple interactions between objects arranged in a time sequence
  + It shows objects with their lifelines and exchange of messages between objects
  + A rectangular line represents an active object. When it is not active, a dashed line is used



**Design Patterns**

* Time-tested framework of organizing object-oriented code to achieve desired functionality
* There are 23 design patterns that can be organized under three broad categories

* + Creational (construct objects by decoupling them from implementing system)
    - Factory
    - Abstract Factory
    - Builder
    - Prototype
    - Singleton

* + Structural (form large object structures between disparate objects)
    - Façade
    - Adapter
    - Decorator
    - Bridge
    - Composite

* + Behavioral (manage algorithms, relationships and responsibilities between objects)
    - Chain of Responsibility
    - Observer
    - Interpreter
    - Mediator
    - Visitor
    - Iterator

* **Abstract factory**: Factory of factories (e.g. email editors)
* **Builder**: Dynamic creation of objects based on easily interchangeable algorithms (e.g. File transfer application)
* **Prototype:** Creates objects based on a template of existing objects through cloning (e.g. in spring beans)
* **Singleton**: Ensures only one instance of the class is allowed within the system (e.g. System class in Java)
* **Façade**: Supplies a simple interface to a set of interfaces within a system (e.g. exposing set of functionalities through a web service)
* **Adapter**: Permit classes with disparate interfaces to work together by creating a common object for interaction (e.g. billing application talking to HR application)
* **Decorator**: Allows dynamic wrapping of objects in order to modify their existing responsibilities and behaviors (e.g. mail sending from business to internal and external users)
* **Bridge**: Defines an abstract object structure independently of the implementation object structure (e.g. JVM talking to underlying OS for a task)
* **Composite**: Facilitate creation of object hierarchies (e.g. displaying info in a shopping cart)

* **Chain of responsibility**: A set of objects should be able to handle a request with the handler determined at runtime (e.g. Exception handling)
* **Observer**: Lets one or more objects be notified of state changes in other objects within the system (e.g. any GUI environment)
* **Interpreter**: Defines a representation for a grammar as well as a mechanism to understand and act upon the grammar
* **Mediator**: Allows disparate set of objects to interact and communicate with each other (e.g. posting in mailing list)
* **Visitor**: Allows one or more operations to be applied to a set of objects at runtime decoupling the operations from object structure (e.g. calculating taxes for different regions)
* **Iterator**: Allows to access the elements of an aggregate object without allowing access to underlying representation (e.g. Java iterator)