# LO6 Prepare UML Models for Software Design

During the design stage, the following tasks are performed:

* Design the user interface and present to the customer for approval
* Develop a high level class diagram(analysis class diagram)
* A detailed class diagram will be created which will detail the exact classes to be created for the software( implementation class diagram)
* Determine how classes will interact with each other to ensure a good software design
* Develop ER diagram from class diagram
* Develop sequence diagrams
* Develop state transition diagrams

## 6.1 Discuss Principles of Graphical User Interface (GUI) Design

* Exercise 6.1
  + The common element in these sites is that they have poor usability.

Usability is defined by 5 components:

1. **Learnability**: Is it easy for users to accomplish basic tasks the first time?
2. **Efficiency**: Once users have learned the design, how quickly can users perform tasks?
3. **Memorability**: how easily can users reestablish proficiency after not using the software for a period of time?
4. **Errors**: How many errors do users make and how easily can they recover from the errors? Are the errors severe?
5. **Satisfaction**: How pleasant is it to use the software?

**What makes a good design?**

1. **Focus on the users and their tasks.** Design the GUI for the intended users. Understand how they perform their jobs and design a GUI that makes it easier for them to perform their job.
2. **Consider Function before Presentation.** It is important to have a good presentation for the GUI but it is more important that it includes all the necessary functionality.
3. **Design from the user’s point of view.** Use language that users understand (avoid computer jargon) and any processes where users have to perform steps that don’t make sense.
4. **Design to make common results easy to achieve.** Consider how often a function must be performed and also by how many users. Frequently used features should require the least button clicks, features required by more users should be more visible. For each function consider:

|  |  |  |
| --- | --- | --- |
|  | **Used by most users** | **Used by few users** |
| **Frequently Used** | Highly visible, few clicks | Barely visible, few clicks |
| **Rarely Used** | Barely visible, more clicks ok | Hidden, more clicks |

1. **Clear and Consistent**. A user has to learn how to use a GUI so there should be no ambiguity. All windows should operate in a similar manner.
2. **Presentation is important (once you know it handles required functionality)**. Consider:

* Visual order and use focus (direct user’s attention to what is important).
* Scannability – avoid long paragraphs, break information into headings, bullets, lists, tables, keep labels short. Users scan looking for what they need.
* Match the medium. If the GUI is to be displayed on a phone it may have different requirements than a large monitor.

1. **Design for responsiveness.** What user values most is speed. Perceived speed is more important than actual speed.

**How do we design a GUI?**

It’s important to note that it’s very hard to change code once it has been written. It’s much easier to change things as early as possible. Client communication – prototype allows for client to see prior to coding.

Prototype: A preliminary model. When we are talking about GUI’s we mean a preliminary view of what our interface might look like.

There are two types of prototyping that can be used during the design process:

1. **Low-fidelity prototypes** - are often paper-based and do not allow user interactions.  They can be hand drawn or a mockup tool can be used.  In theory, low-fidelity sketches are quicker to create. Low-fidelity prototypes are helpful in enabling early visualization of alternative design solutions, which help provoke innovation and improvement. An additional advantage to this approach is that when using rough sketches, users may feel more comfortable suggesting changes.

Optional materials for designing prototypes are:

* + Paper and pencil
  + Whiteboard
  + Flipchart
  + Sticky notes
  + Mockup tools: <http://www.balsamiq.com/products/mockups>

1. **High-Fidelity prototypes** - are computer-based, and usually allow realistic (mouse-keyboard) user interactions. High-fidelity prototypes take you as close as possible to a true representation of the user interface. High – fidelity prototypes are assumed to be much more effective in collecting true human performance data (e.g. time to complete a task), and in demonstrating actual products to clients, management, and others. “Wizard of Oz” techniques are often used.
   * https://www.nngroup.com/articles/ux-prototype-hi-lo-fidelity/

Usually you want to start with the low fidelity and test, make changes (and assuming the changes aren’t huge) move over to a high fidelity prototype.

**Ordering of Screen Data and Content**

* Divide information into units that are logical, meaningful, and sensible.
* Organize by the degree interrelationship between data or information.
* Possible ordering of controls include:
  + Conventional. (how do others usually do this?)
  + Sequence of use. (ie. The order of the steps involved)
  + Frequency of use.
  + Function.
  + Importance.
* Form groups that cover all possibilities.
* Ensure that information that must be compared is visible at the same time.
* Ensure that only information relative to the users’ tasks or needs is presented on the screen.

**Screen Navigation and Flow**

* Locate the most important and most frequently used elements or controls at the top left.
* Maintain a top-to-bottom, left-to-right flow.
* Assist in navigation through a screen by:
  + Aligning elements.
  + Grouping elements.
* So in the early 20th Century a group of German psychologists developed a number of theories to explain how visual perception works. These theories are known as the Gestalt principles of visual perception (Gestalt is the German word for shape). Some of these principles can actually be applied to user interface design.
  + **Proximity**– items that are near each other appear grouped. You can keep a clean design clean of group boxes and boarders simply by visually grouping items by use of space. This also applies to labels which should be in close proximity to the controls they describe.
  + **Similarity** – items that look similar appear to be grouped. So if you have a number of text boxes that are the same size and proximity to each other, they will appear grouped.
  + Using line borders.
* Through focus and emphasis, sequentially, direct attention to items that are:
  + - Critical.
    - Important.
    - Secondary.
    - Peripheral.
* Tab through window in logical order of displayed information.
* Locate command buttons at end of the tabbing order sequence.
* When groups of related information must be broken and displayed on separate screens, provide breaks at logical or natural points in the information flow.

## Resources:

Usability Guidelines:

<http://guidelines.usability.gov/>

Guidelines for creating websites:

<http://static.lukew.com/webforms_lukew.pdf>

General form design:

Book from books 24x7: Step 3 - Understand the Principles of Good Screen Design: Human Considerations in Screen Design.

<http://www.books24x7.com/toc.aspx?bookid=4883&chunkid=502518253&rowid=228&refid=T71VE>

**How can I test my design to decide if it is good?**

* **Heuristic Evaluations** – when a usability expert reviews the interface and compares it against accepted usability principles.   
  <http://www.usability.gov/how-to-and-tools/methods/heuristic-evaluation.html>
* **Usability Testing** – Evaluating by testing the product with representative users. Typically, participants try to complete tasks while observers watch. The goal is to identify usability problems, collect qualitative and quantitative data to determine participant’s satisfaction of the product. Can use the paper prototypes to test, and ask users where they would click.

Chalkmark – testing GUI - <http://www.optimalworkshop.com/chalkmark.htm>

**My design is done. Now what?**

The client must approve the designs.

## 6.2 Describe Process for Extracting Classes and Methods from Specifications

**Construct classes from user stories**

One more task that we do during the analysis phase is to create a “class diagram” of our system. Typically, this is done after you’ve created your use case diagrams, scenarios, activity diagrams, and user stories, but when you are creating your class diagram, there will be additional things that you notice that will require you to update your other analysis models.

You are probably familiar with class diagrams from your Java class, but at the analysis level, rather than using the diagrams to describe the classes that you will be writing in your program, you will be describing real-world classes and interactions, classes that are meaningful to the customer. This diagram helps you and your client to understand the system that you are building as a whole. You will likely use the analysis class diagram as the basis for your program design, but that will contain many more classes that are not part of the conceptual view of the system (ex. Button listener!). This more detailed diagram is called the “implementation class diagram”, and we’ll look at it as well.

Remember:

* **Analysis Class Diagram** 🡺 helps you understand the system and what it is to do
* **Implementation Class Diagram** 🡺 the design of your program

Quick Review of Object-Oriented Terminology

* We model the system using "objects".
* In the object oriented view of software the software is developed around objects.
* Each object is responsible for its own data and its own functions.
* In the object oriented view of software data and code is broken down into components called objects.
* An object represents an entity it may be physical or abstract.
* Examples of objects in the real world are: people, buildings, vehicles, windows, trees, books.
* An object may contain other objects:
  + I.e. a building object can contain wall objects, door objects, etc.
* Each object is responsible for its own data and its own behaviour or functions.
* How do objects relate to classes?
  + Classes defines the object and an object is one instance of that class.
  + For instance: a blueprint specifies all the building details for a house but multiple buildings can be built from a single blueprint. The blueprint is like the class and each house built is like an object.
  + Another example is the person class. The person class defines certain behaviors and attributes and then each object is an instance of that person class with specific values for the attributes and behaviors.
  + Example: “Person” could be a class, and Coralee and Jason could be objects of that class. Coralee is an instance of the Person class, as is Jason.

Look at the following terms: Dog, Rover, and Spot. One of these is a class, and the other two are objects of that class.

*Which are the objects?* Rover and Spot

*What is the class?* Dog

**UML Modeling Techniques**

* UML modeling techniques focus on an object oriented approach to software development
* During the analysis phase the initial breakdown of the project into classes is considered.
* At this phase think only of the high level classes and not all the classes that will be required for the complete implementation of the system.
* During the design phase an analysis class diagram will be created that can be used during implementation.
* Using an object oriented approach to software development is very important when developing code in an object oriented language like Java or C#.

**Identifying the Classes**

* There are many ways to identify classes from the project documentation created so far.
* One method is to highlight all the nouns from the use case diagrams and user scenarios. Once this list is created, analyze it to determine which nouns would be appropriate classes.
* Review the use case diagrams and scenarios for verbs. Verbs tend to indicate behaviours or functions that need to be included in the system. The ones that must be included in the system will become behaviours of the individual classes so at this point attempt to assign them to classes as well.
* Adjectives and adverbs can be used to help identify the attributes of the objects. These attributes describe the state of an object at any point in time and it is likely they might need to be represented as data in the object.

*Using the example of logging into Windows, what would be three classes and the attributes and behaviours of those classes?*

Here are some possible answers (not necessarily the correct answers):

* User: username, password
* Windows: login screen
* Domain controller: list of valid username / password pairs, validate password

## 6.3 Develop Analysis and Design Class Diagrams

**Develop analysis class diagram**

* Once the classes have been identified they can be modeled in a UML class diagram.
* Keep in mind Class Diagrams should be a high level view and will be expanded on later.
* The analysis class diagram is intended to give a basic idea of entities and high level functionality to be included in the system.

**Analysis Class Diagram**

* The analysis class diagram represents classes and relationships between classes. A class is represented using the following symbol:



* For the purpose of the analysis class diagram relationships ("associations") between classes will be shown by a line linking the two classes. Label the association with a description of how the one class interacts with the other class. Example: two classes, Person and Door. The relationship is that the Person “opens” the door.
* Attributes associated with classes are the data that is associated with that class.   
  *For example, a person may have a name, address, and phone number as attributes.* Attributes are implemented in the code as **instance variables**.
* The behaviours are functions performed by that class.
* A class is responsible for implementing all of its own behaviours.

*For example, a person class could have behaviours like talk, walk, and comb hair as functions it can perform. Functions are implemented in the code as* ***methods****.*

For the analysis class diagram do not be too concerned about the attributes and behaviours of the class. If any are known they can be added at this point but the analysis class diagram is a high level look at the class so it is not necessary to include all the details of attributes and behaviours.

I like to draw the classes (usually on a whiteboard), then step through the scenarios, ensuring I've got classes to represent everything in the scenario. Anytime one class needs to interact with another class, I add an association between them, labelling the association with what the interaction is.

**Describe Implementation Class Diagram**

### Implementation Class Diagram

* Analysis class diagram identifies general classes required, at a high level.
* The analysis class diagram is reworked and expanded during design to identify all required classes and public methods.

### Creating an Implementation Class Diagram

* During the design phase, it is necessary to determine all the classes that must be implemented in the code. The following functions must be completed:
  + Determining the actual classes that are required.
  + Determine the responsibilities of each class.
  + Determine how the classes will interact with each other.

Differences between the analysis class diagram and implementation class diagram:

* Methods involved are included in the Implementation class diagram.
* Relationships are defined in the Implementation class diagram.
* Implementation Class diagram is more comprehensive (listing of all classes).

### Association Between Objects

* Objects can have associations (or relationships) with each other. Identifying these relationships will help increase understanding of the system and how it should be implemented. These associations are represented in the class diagram.

There are four different kinds of relationships:

* Inheritance
* Composition
* Aggregation
* Uses (Association)

#### Inheritance Relationship

* In inheritance one class is a parent and one class is a child.
* The child class inherits all the attributes and behaviours associated with the parent.
* The child class can add attributes and behaviours specific to the child.
* An inheritance relationship is indicated by an open triangle that points from the child to the parent.



**Person Object:**

ID234323  
Bob Brown  
306-222-2222  
123 here, Saskatoon

**Staff Object:**

SM212121  
Jane Doe  
306-333-3333  
321 there, Saskatoon  
ICT

**Customer Object:**

SM212121  
John Doe  
306-333-3333  
321 there, Saskatoon  
01/01/2014

In the example above, Customer and Staff inherit from person. So we can say that the customer is-a-kind-of person and Staff is-a-kind-of person.

#### Aggregation Associations

##### Whole-part Relationships

##### Aggregation

Aggregation depict classes that are made up of smaller classes.

* Weak form of aggregation.
* One class can still exist without the other. Ask yourself, what would happen if one of the classes in the relationship was taken away? If it can exist, it is an example of aggregation.
  + Example: A car that has a driver is still a car if the driver leaves the car. A driver still exists if they leave the car.
* Identified by a hollow diamond which touches the whole while the other class is the part.



* Consists-of, has-a, is-part-of

##### Composition

* Strong form aggregation.
* In a strong association, one class cannot exist without the other class or classes. The lifetime of the part is bound within the lifetime of the whole.  
  + Example: A house with several rooms including a kitchen would be considered incomplete if the kitchen was removed. The removed kitchen would likely not separately exist after being removed from the house.
* Identified by a filled in diamond which touches the whole while the other class is the part.



#### Uses (Association)

* A relationship that is used for relationships that are not inheritance, aggregation, or composition.
* Identified by an arrowhead that point from the source to the target object. Can also think of this as the class that acts as a client uses or depends on another class that acts as a supplier. The arrow is directed from the client to the supplier.



Example:

School has a collection of departments. The school can’t exist without the departments and vice versa so it an example of composition. Departments are made up of instructors. An instructor can still exist without a department and vice versa so it is an example of aggregation. (Students are similar) Departments provide classes (use). Courses are assigned (use) Instructors and Students.



## Develop Implementation Class Diagram from Analysis Diagram

|  |  |
| --- | --- |
| **Relationship Type** | **Relationship description** |
| Inheritance: Instance of a class (object) | Is-a |
| Inheritance: Class that inherits from another class | Is-a-kind-of |
| Aggregation/composition: Whole (container) | Consists-of, has-a |
| Aggregation/composition: part | Is-a-part-of |
| Association(Uses): Classes are associated | Uses or another description of the relationship  (Example: logs in) |

## Details of Class Model Nouns: classes Verbs: methods Adjectives/Adverbs: Attributes

When the implementation class model is created, consideration should be given to actual data types and methods required for each class.

### Attributes

The attributes will be stored as data in a class so the type of data storage required is indicated in the implementation class diagram. The following format is used:

-numSides : int

Data type

Separator

Variable name

Access modifier

* The access modifier identifies the visibility of the attribute or method outside of the class. It can have the following visibilities:
  + + public – accessible by all other classes
  + # protected – accessible by the class and all child classes
  + – private – accessible by the class only

Example:

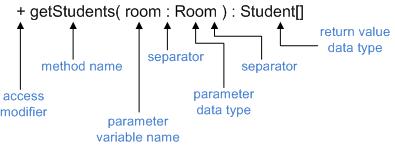
|  |
| --- |
| **BasicBox** |
| -length: int  -width: int  -height: int |
|  |

### Methods

Implementation class diagrams include:

* Methods
* Parameters
* Return values

The following format is used for method declaration in a class diagram:



**Example:**

|  |
| --- |
| **BasicBox** |
| -length: int  -width: int  -height: int |
| +Make(l, w, h : int)  +setLength(l : int)  +setWidth(w : int)  +setHeight(h : int)  +volume : integer  +out : string |

## 6.4 Plan Class to Database Schema Conversion

Most programs require a database so we need to look at converting our class diagram to a database design. The database design is recorded in an Entity-Relationship model (E-R model). The E-R model is a model that has been used for way longer than object-oriented design; we'll convert our UML implementation class diagram into an E-R model.

You'll also hear the E-R Model called the E-R diagram (or ERD).

### Database Terminology Overview

A database allows permanent storage of data in an organized and easily accessible manner. A database stores data in a series of tables. Each table is a series of row and columns similar to an Excel spreadsheet. The following information is important to know about databases.

* A **database** comprises a number of tables to store data.
* A **table** comprises an arrangement of data in columns and rows.
* The columns of data are called **fields.** Each field holds data for a specific type of information. A table may have one or more fields.
* The rows of data are called **records.** Records are similar to rows in an Excel spreadsheet. A table may have one or more records.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Student Number | First name | Last name | Address | Phone number |
| 123456789 | Shane | McDonald | 123 Fake St. | 306-555-1234 |
| 987654321 | Rob | McDonald | 404 Error St. | 306-555-4321 |
| 456789321 | Sharon | McDonald | 123 Fake St. | 306-555-1234 |

* A **primary key** is the field in a table that uniquely identifies each row in the table. A primary key must be unique for the table, meaning it only maps to exactly one row in a table.

Consider the following table: *Which of these fields uniquely identifies a student?* None! More than one student can have the same first name, last name, address or phone number.

|  |
| --- |
| Student |
| First name |
| Last name |
| Address |
| Phone number |

*What could we use to uniquely identify a student?* Student number – no two students have the same student number. The new table would look like this:

|  |
| --- |
| Student |
| First name |
| Last name |
| Address |
| Phone number |
| Student number |

Lots of times, there won't be a unique field. In that case, you can arbitrarily create a unique identifying number, and make sure you never assign the same number to more than one row of the table.

In some cases, more than one field combines to create a primary key for the field. Suppose we have another table that keeps track of student marks.

|  |
| --- |
| Student Mark |
| Student number |
| Course number |
| Mark |

We can have more than one row with the same student number (students can take multiple courses), and we can have more than one row with the same course number (more than one student can take the same course). We can even have multiple students with the same mark! But, there will be only one row with a combination of student number & course number (assuming a student only takes a course once). The combination of student number & course number uniquely identifies a row, so those two fields combined act as the primary key. This is called a **composite key.**

A **foreign key** is a field used in one table that uniquely identifies a row in another table. Not all tables have a foreign key.

*Is there a foreign key in the Student table? If so, what is it?* No.

*Is there a foreign key in the Student Mark table? If so, what is it?* Yes, the student number field in the student mark table is a foreign key – it uniquely identifies a row in the Student table.

One of the things you'll learn about in the 2nd year database class is "database normalization" – you define your database structure so that you don't have redundant data in your tables. In that case, you have lots of foreign keys in your tables, and relationships between tables. Associations between classes are similar to relationships between database tables!

### Entity Relationship(ER) Model Overview

The entity relationship (ER) model is used to graphically model the structure of a database. To be able to create a database the following information must be known:

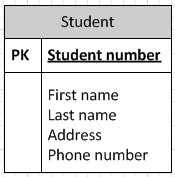
* The number of tables in the database
* The name of each table in the database
* The number of fields in each table
* The names of each field in each table
* The primary key for each table
* Any foreign keys

This information is available from the ER model.

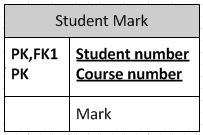
## Components

An entity-relationship diagram has two components.

* **Entity.**  A person, place or thing about which multiple instances of data must be stored. This will be represented as a table in the database. The entity will have a name, which will be the name of the table. Attributes associated with the entity are also indicated. This will be the data that must be stored for each instance of the entity. These attributes are represented as fields in the database. The entity is modeled as:



A primary key is identified in the table by the PK designation. If a foreign key is present it is identified by the FK designation. The Student Mark table has both a composite key AND a foreign key.



* **Relationships.**  Relationships between tables define cardinality (to be explained) when explaining how the tables relate to each other. Three symbols are used to represent cardinality:
  + ring represents zero: O
  + dash represents one: |
  + crow’s foot represents more than one : 

## Four types of Cardinality

These symbols are used in pairs to represent four types of cardinality that an entity may have in a relationship.

* Ring and dash – zero or one

ZeroOrOneCropped.JPG

* Dash and dash – exactly one

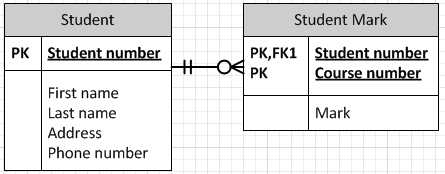
ExactlyOneCropped.JPG  
ring and crow’s foot – zero or more  
ZeroOrMoreCropped.JPG

* Dash and crow’s foot– one or more  
  OneOrMoreCropped.JPG

*Think about the Student and Student Mark table. For every entry in the Student table, how many entries will there be in the Student Mark table? Choices: 0 or 1, exactly one, zero or more, one or more?*

* A student might not have taken any classes, or they could take many classes: zero or more.

Here is an ERD for the Student and Student Mark tables:

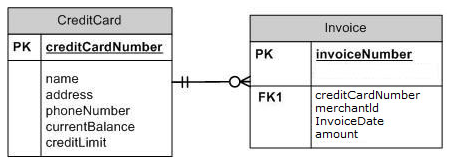


**Practice:** Given the following two tables related to credit card transactions, draw an ERD (just do it on paper). Start by identifying the primary key, then any foreign keys (note: there isn't always a foreign key).

|  |
| --- |
| Credit Card |
| Name |
| Address |
| Phone number |
| Credit card number |
| Current balance |
| Credit limit |

|  |
| --- |
| Invoice |
| Invoice date |
| Invoice number |
| Merchant ID |
| Amount |
| Credit card number |

**Solution**:



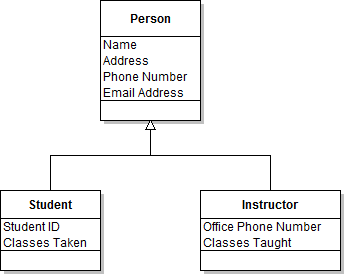
## 6.5 Develop ER Model from Class Diagram

Most programs require a database so we need to look at converting our class diagram to a database design. The database design is recorded in an entity relationship diagram. (ER model or ERD)

The class diagram should be evaluated for classes that require permanent storage. Generally one database is created for each class that needs permanent data storage. If a class should be a table in the database – most (possibly all) attributes of the class will become fields in the database table.

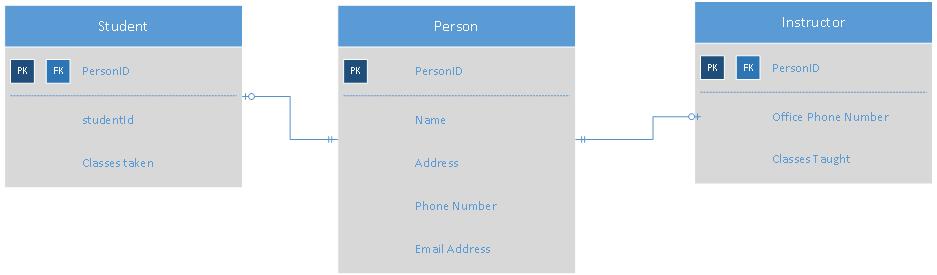
## Converting Classes using Inheritance into an ER diagram

Consider the following class diagram:



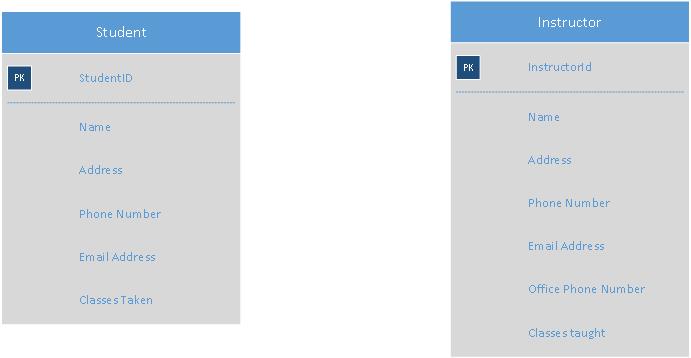
All three of these classes contain data that should be stored in the database. When inheritance is involved there are three different approaches to storing the data in a database.

1. Create one DB table per class. Create one table for Person, one for Instructor and one for Student. This means if we want instructor information we must retrieve a record from the Person table and the Instructor table.



This approach has impacts on performance since we must do multiple DB table lookups to retrieve the information for instructor or student. This approach does offer the most flexibility.

1. Create one table for each child class. The parent’s attributes are repeated in the child classes.

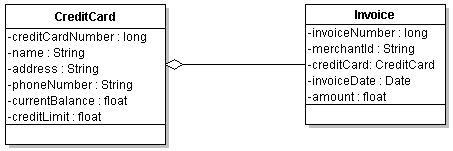


This approach has performance gains since we only have one database lookup when accessing student or instructor. But maintenance will increase if we have to change the attributes associated with the Parent class – since it now needs to change in two places.

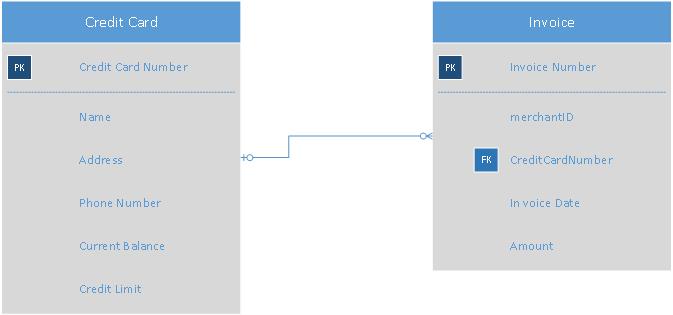
1. Create a single table for all classes in the hierarchy. This table would contain attributes for Person, Student AND Instructor. This option means for some rows, some fields will be blank. For a student in the table all instructor related fields will be blank. This approach offers ease of accessing the table and easy maintenance but it wastes space.

## Converting Classes that use Composition and Aggregation

Classes that use aggregation or composition can each be translated directly into their own table. The tables will be linked with foreign key.



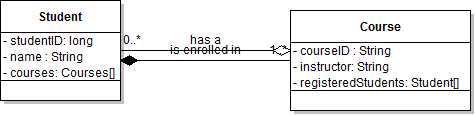
In this example create a table for CreditCard and a table for Invoive. The credit card number will be a primary key in the credit table and a foreign key in the Invoice table.



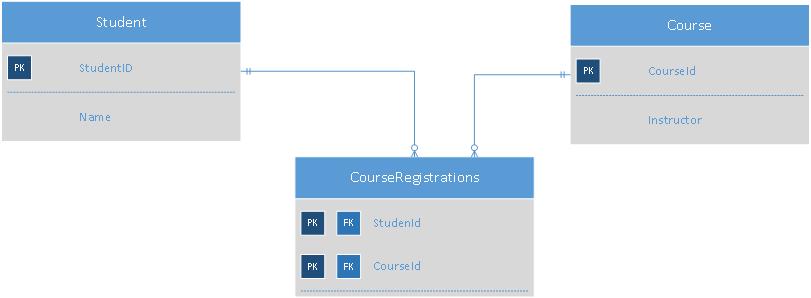
Note: If the relationship between the two classes in aggregation and composition will be an exactly 1 to 1 relationship likely you want to create only one database table. (For example, Person and Address in employment agency example)

### Converting classes that have a many to many relationship

In some cases we have classes that have a many to many relationship that must be represented in the database. Consider the following example:



A student can take any number of classes and a course may have any number of students registered. We cannot effectively store multiple values in one field of a database table. To solve this many to many problem we create an additional table (called a flattening or joining table) that maps students to courses.



A student will have one record in the Course Registrations table for each course they are taking. Similarly, we can find all the students in a particular course by searching the CourseRegistration table.

**Student Practice (Exercise 6.5)**: Individually work to convert a class diagram to ER diagram. Use the AnimalFarmPractice diagram in the Exercise 6.5 folder on the Handout drive.

## 6.6 Describe Sequence Diagram

While an implementation class diagram shows you what classes make up your software, it doesn’t show how the classes work together. For this you use a sequence diagram which depicts the order (or sequence) of events at runtime. You can think of it as a pictorial representation of your code.

**Sequence diagrams**

* A UML diagram used to describe interactions between objects in a software system.
* Clearly shows the sequence (order) of messaging between objects.
* Helps developers to understand and visualize the flow of logic between objects within the system.
* Extends information from the class diagram by showing interactions between instances of classes (OBJECTS).

The main reasons for creating sequence diagrams are:

* Increase understanding of the system before writing code. Better understanding of the system before code is written will greatly reduce the amount of code rework necessary.
* Better understanding of the whole system. Sequence diagrams help all developers on the project understand how the system works from a higher level. This would be very difficult to do by looking just at the code.
* Improving testing and execution. Sequence diagrams allow the developers to test execution of the code before actually writing the code.

**NOTE:** It is important to understand that it is not usually necessary, or even a good idea, to attempt to create sequence diagrams for every aspect of the system you are building. Sequence diagrams are most useful for helping you to work through complicated subsystem or bit of logic; creating sequence diagrams for simple, straightforward use cases or actions would be pointless and wasteful.

* You will take your class diagram and by using the elements from it (ie: classes, attributes, methods) you will create the Sequence diagram.

**NOTE:** The models you create as part of the design phase, and any phase of your project, are living and should evolve as your understanding of the system changes. You may, when creating the sequence diagram, identify an area where a set of classes could be broken down further to simplify interactions in the system; if this happens, you should flow that update through **ALL** of your diagrams so that the change is documented for future users.

* IMPORTANT: Your sequence diagram should not have any classes, attributes (parameters), methods that do not show up in other documentation, specially your class diagram. If you find that as you go through creating the sequence diagram that you are **missing** a method or class, go and **add** it to the class diagram.

You’d typically create a sequence diagram to model the following:

1. Usage scenarios. How the system will be used and/or interacted with. May model part of, or an entire use case or sequence of use cases.
2. Method logic. Help in breaking down complex logic.
3. Service logic. Effective documentation for services which are intended to be consumed by other systems.

**NOTE:** Sequence diagrams are probably the most useful diagrams for day-to-day development because of their being closer to the implementation and logic of the system.

### Sequence Diagram Components

**Targets**

Targets are the parts of the sequence diagram whose interactions are being modeled. They can be either actors (*which you would have identified during the creation of your use case diagram*) or objects.

Actors may be represented as a stick figure, underneath of which is the name of the actor. In newer diagrams using UML2, you may see actors noted in the same way as an object, but with a stereotype of <<Actor>>. Actors will normally be placed in the far left gutter of your interaction diagram.



Guest

<<Actor>>

Objects are represented as a rectangular box with the normal UML notation for object names inside. An object with only a class name specified would look like this:

: Student

An object with both a class and object name (instance name) would look like this:

enrollingStudent : Student

**NOTE:** the formatting: the object name, a separator and the class name, identical to the formatting for attributes in a class diagram.

It isn’t normally necessary to include the object name unless you’ll need to specifically reference it later, or if you may have more than one instance of that class.

**Lifelines**

A life line indicates the existence of an object or actor. It is represented by a dashed line extending down from the object or actor whose lifeline is being modeled, terminating with an “x” where the lifetime of the object ends (where it is destroyed/freed). Making the end of the lifeline of an object can be useful if you are using a language that requires manual memory management.



**Activations**

An activation represents the length of time a method class is active. It is indicated as a box on the lifeline. Remember that the activation exists from the point of the initial method call to the time it exits (returns a value, continues). There may be many activations on a single object lifeline.



**Message**

A message represents the communication between class instances in your sequence diagram. They’re represented as lines either between individual object lifelines, or from an object lifeline to itself. Different types of messages are noted with different types of arrow heads. The arrow will **always** point in the direction of the method call (*from caller to callee*).

There are several specific types.

1. A solid line indicates that a message is being passed or a method is being called. The message includes the name and parameters of the method being called.

**NOTE:** *Synchronous messages are far the most common way to pass messages.*

* 1. A solid arrow head at the end of the line indicates a synchronous message, which means the caller waits for a reply before continuing with its own processing.

Method(parameter)

* 1. An open arrow head at the end of the line indicates an asynchronous message, which means the caller does not wait for a reply before continuing with its own processing. **NOTE:** Asynchronous messages will be common if you are working with threads or some other form of multiprocessing system.

Method(parameter)

1. A dashed or dotted line with an open arrow head indicates a return value from a message. The response value can be indicated on the line. It is not necessary to always include the return value if it’s obvious (Ex. Noting the return value of getName), but it can be a good practice to get in to if your diagram won’t end up being too cluttered.

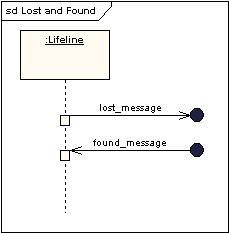
returnValue

**Tips:**

* Messages and return values can go in either direction, which may not immediately be obvious; as the ordering of the object lifelines is not defined, the communication between objects doesn’t follow a defined order.
* Messages between objects are composed of the signature of the method being called along with parameters, possible parameter values, and possibly a label for the stereotype of the method (Ex. <<contruct>>).
* Messages between Actors and objects take the form of a short phrase describing the interaction, such as “Student chooses a class” or “User attempts to log in”.

**Lost and Found**

There may be cases where you spread the description of the system you are modeling across multiple diagrams, and need to describe the points where the flow of control moves in and out of each diagram. You can do this using messages pointing to or coming from a single black circle (the same you use to identify the starting point of a use case diagram).



The arrow pointing to the black dot means that the message is intended for a recipient not shown on the current diagram, and the arrow originating from the black dot means that the message is coming from a sender not identified on the current diagram.

**Self messages**

Objects will frequently send messages to themselves**.** Ex.: accessing getters and setters, helper methods.

You can model these with a set of stacked activation boxes, each subsequent one offset slightly to the right, with the same rules for notating the message lines.

## Frames – Alternatives, Options and Loops

Frames - regions or fragments of the diagram used to support conditional and looping constructs.

Guard expressions - logical statements (conditions) placed between square brackets indicating what needs to be true in order for the messages to be sent.

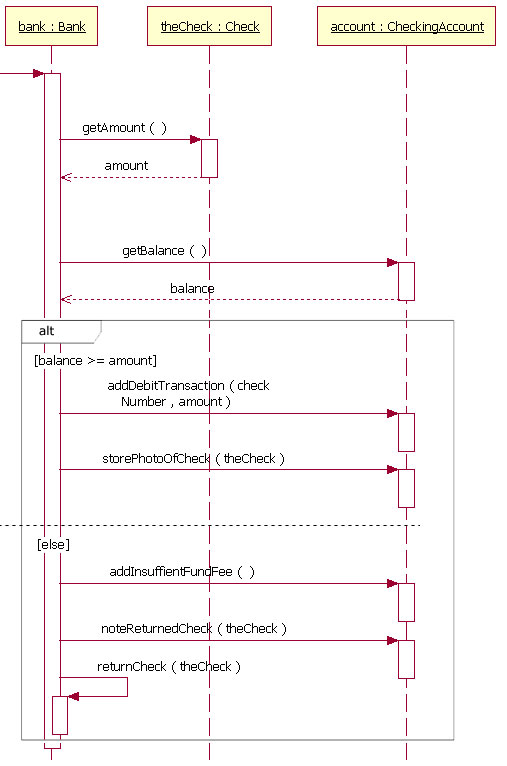
Is drawn as a box surrounding the messages performed as part of the conditional or loop. Has a title in the upper left corner of the box (sometimes referred to as an operator) describing whether it is a conditional or loop.

Within the frame, guard expressions are placed above the applicable messages.

**Alternatives**

Alternative fragment is used for conditionals that have mutually exclusive logic.

On the sequence diagram, you will draw a frame and label it “alt”. Remember to place the guard expressions above the applicable messages.



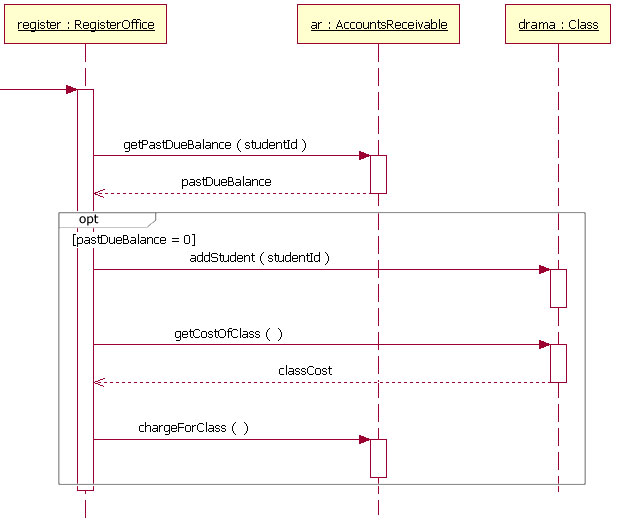
Ex. - in the case of a payment processing system, you might execute one series of methods if the user has sufficient funds to complete the transaction, but another set of methods if they did not.

In the diagram above, the first subsection of the alternative fragment will evaluate only if the balance available is able to cover the amount owing, otherwise the second subsection will be evaluated.

**Options**

Optional fragment is used for logic where certain messages are only sent when a certain condition holds true; similar to an alternative, but where there is no alternative option.

On the sequence diagram, you will draw a frame and label it “opt”. Remember, to place the guard expressions above the applicable messages.



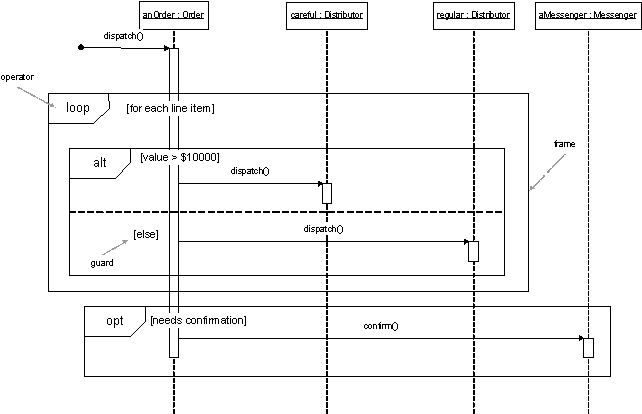
In the example above, the messages in the frame will only run if pastDueBalance =0

### Loops

**NOTE:** Loops are marked up in a way very similar to alternatives and options.

A frame marked with “loop” is created to enclose all messages which will take place in the loop.

A guard expression is placed which holds the conditions under which the loop will continue to run. Examples: “[ moreLinesToProcess = true ]”, or “[ for each line ]” . Does not matter the wording as long as the intent is obvious.



In the above example, the loop will be executed for each line item in an Order. Notice how the loop fragment has an alternative fragment inside. So the loop will execute the code inside the alternative fragment IF it meets the condition.

<https://www.bing.com/videos/search?q=umlsequence+diagram+tutori&view=detail&mid=7DAA4A5B6CFF9626BC4A7DAA4A5B6CFF9626BC4A&FORM=VIRE> (Approx. 12 minutes)

## 6.7 Develop State Transition Diagram

<http://www.tutorialspoint.com/uml/uml_statechart_diagram.htm>

<http://code.tutsplus.com/tutorials/how-using-state-diagrams-can-make-you-a-better-web-coder--net-33>

<http://www.stickyminds.com/article/state-transition-diagrams>

State diagrams are UML diagrams that are used:

* To model object states of a system and/or object.
* To model reactive system. Reactive system consists of reactive objects.
* To identify events responsible for state changes.
* For forward and reverse engineering.

State transition diagrams can be useful when designing a class. They can help identify attributes and behaviours of the class. They can be useful in modeling the user interface. The various states that a window can be in can be detailed. For example, some fields may be greyed out in certain states.

The following components are used in a state transition diagram:

* **Initial State**.
  + A starting state.
  + Indicated by a filled in circle.

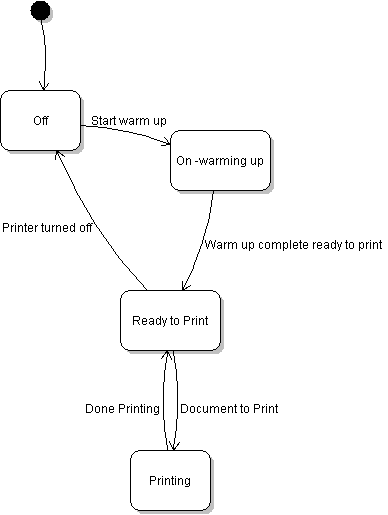


* **Final State**.
  + A final sate.
  + Indicated by a circle with a hollow border.  
      
         
* **State**.
  + Condition of an object at a moment in time (time between events).
  + Indicated by a rectangle with rounded corners.  
      
    
* **Event**.
  + A significant occurrence that triggers moving from one state to another state.
  + Indicated by an arrow between states. The event is indicated by text associated with the arrow.

Example: a telephone



Example



Above is a sample state diagram for printer.