**Logo, company name

Description automatically generated**

**COMSATS University Islamabad (CUI)**

Software Design Description   
(SDS DOCUMENT)

for

**Project Title**

Version 1.0

***By***

**Student Name 1 CIIT/SP18-BCS-xxx/ISB**

**Student Name 2 CIIT/SP18-BCS-xxx/ISB**

***Supervisor*Supervisor Name**

***Co-Supervisor (if any)*Co-Supervisor Name**

*Bachelor of Science in Computer Science (20xx-20xx)*

**Table of Contents**

Revision History iii

Application Evaluation History iv

1. Introduction 1

2. Design Methodology and Software Process Model 1

3. System Overview 1

3.1 Architectural Design 1

4. Design Models 1

5. Data Design 2

5.1 Data Dictionary 2

6. Human Interface Design 2

6.1 Screen Images 2

6.2 Screen Objects and Actions 2

7. Implementation 3

7.1 Algorithm 3

7.2 External APIs/SDKs 5

7.3 User Interface 5

7.4 Deployment 8

8. Testing and Evaluation 9

8.1 Unit Testing 9

8.2 Functional Testing 9

8.3 Business Rules Testing 10

8.4 Integration Testing 10

Appendix A 11

Appendix B 13

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Date** | **Reason for changes** | **Version** |
|  |  |  |  |
|  |  |  |  |

Application Evaluation History

|  |  |
| --- | --- |
| **Comments (by committee)**  **\*include the ones given at scope time both in doc and presentation** | **Action Taken** |
|  |  |
|  |  |

Supervised by

Supervisor’s Name

Signature\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Introduction

Briefly explain scope of the project covered till now including modules.

# Design Methodology and Software Process Model

Explain and justify the choice of design methodology being followed. (OOP or Procedural). Also explain which process model you are following and why.

# System Overview

Give a general description of the functionality, context, and design of your project.

Provide any background information if necessary.

## Architectural Design

Develop a modular program structure and explain the relationships between the modules to achieve the complete functionality of the system. This is a high-level overview of how the system’s modules collaborate with each other in order to achieve the desired functionality.

Don’t go into too much detail about the individual subsystems. The main purpose is to gain a general understanding of how and why the system was decomposed, and how the individual parts work together.

Provide a diagram showing the major subsystems and their connections.

* In initial design stage create Box and Line Diagram for simpler representation of the systems
* After finalizing architecture style/pattern diagram (MVC, Client-Server, Layered, Multi-tiered) create a detailed mapping modules/components to each part of the architecture

To view example of box and line diagram and architecture styles, see Appendix A.

# Design Models

Create design models as are applicable to your system. Provide detailed descriptions with each of the models that you add. Also ensure visibility of all diagrams.

***Design Models for Object Oriented Development Approach***

The applicable models for the project using object oriented development approach may include:

* Activity Diagram
* Class Diagram
* Sequence Diagram
* State Transition Diagram (for the projects which include event handling and backend processes)

***Design Models for Procedural Approach***

The applicable models for the project using procedural approach may include:

* Activity Diagram
* Data Flow Diagram (data flow diagram should be extended to 2-3 levels. It should clearly list all processes, their sources/sinks and data stores.)
* State Transition Diagram (for the projects which include event handling and backend processes)

To view examples of all above models, see Appendix B

# Data Design

Explain how the information domain of your system is transformed into data structures. Describe how the major data or system entities are stored, processed, and organized.

List any databases or data storage items.

## Data Dictionary

Alphabetically list the system entities or major data along with their types and descriptions. If you provided a functional description, list all the functions and function parameters. If you provided an OO description, list the objects and its attributes, methods and method parameters.

# Human Interface Design

Describe the functionality of the system from the user’s perspective. Explain how the user will be able to use your system to complete all the expected features and the feedback information that will be displayed for the user.

## Screen Images

Display screenshots showing the interface from the user’s perspective. These can be hand-drawn, or you can use an automated drawing tool. Just make them as accurate as possible. (Graph paper works well.)

## Screen Objects and Actions

A discussion of screen objects and actions associated with those objects

# Implementation

This chapter will discuss implementation details of the project. You will not put your source code here, however, are required to write the core modules functionalities in pseudocode form (Following sections are required in this chapter).

Note: You are required to follow proper coding standard to write your source code. For guidelines, **General Coding Standards & Guidelines** are provided in Appendix D.



## Algorithm

Mention the algorithm(s) used in your project to get the work done with regards to major modules. Provide a pseudocode explanation regarding the functioning of the core features. Be sure to use the correct syntax and semantics for algorithm representations. Following are few examples of algorithms/pseudocode:

|  |  |  |
| --- | --- | --- |
| **Algorithm 1 MHCF co-authorsBasedClustering** | | |
| **Input:** n groups Gn where each group has set of papers (pr) | | |
| **Output:** Set of system generated clusters/groups Gn | | |
| 1: merge ← true  2: Flag ← false  3: **While**(merge==’true’) **do:**  4: merge ← false  5: **for** i in range (0: len(G)-1):  6: **for** j in range (i+1: len(G)):  7: **if** (similarCoauthors(GiLco-authors, GjLco-authors) == true) **then**  8: Flag ← true  9: **Else** (checkNameFragments(GiLco-authors, GjLco-authors) == true) **then**  10: Flag ← true  11: **if** (Flag == true) **then**  12: Gi ← Gi U Gj  13: G ← G.pop(j)  14: merge ← true  15: **end if**  16: **end for**  17: **end for**  18: **end while** | | |
| **Algorithm 1.1 checkNameFragments** | | |
| **Input:** two lists GiLco-authors, GjLco-authors where each list has co-authors | | |
| **Output:** Boolean value | | |
| 1: Flag ← false  2: Count ← 0  3: **foreach co-author1 in GiLco-authors:**  4: coauthor1Fragments ← co-author1.split(‘ ’)  5: **foreach co-author2 in GjLco-authors:**  6: coauthor2Fragments ← co-author2.split(‘ ’) //split author name into name fragments  7: **if** (len(coauthor1Fragments) == 3 **and** len(coauthor2Fragments) == 3) **then** //both authors have three name fragments  8: **if** (len(coauthor1Fragments[0]) ≥ 3 and len(coauthor1Fragments[0]) ≥ 3) **then**  9: //both authors first name have more than three characters  10: **if**(coauthor1Fragments[0] == coauthor2Fragments[0]) and (coauthor1Fragments[2] == coauthor1Fragments[2])) **then**  11: //both authors have same full first and last name  12: **if** ((coauthor1Fragments[1] == coauthor2Fragments[1]) or (coauthor1Fragments[1][0] == coauthor2Fragments[1][0])) **then**  13: //both authors have same middle full name or same first character of middle name  14: Count++  15: **end if**  16: **elseif** ((coauthor1Fragments[0][0] == coauthor2Fragments[0][0] and coauthor1Fragments[0][1] == coauthor2Fragments[0][1] and coauthor1Fragments[0][2] == coauthor2Fragments[0][2]) and (coauthor1Fragments[2] == coauthor1Fragments[2])) **then**  17: //both authors have same first three characters of first name and full last name  18: **if** ((coauthor1Fragments[1] == coauthor2Fragments[1]) or (coauthor1Fragments[1][0] == coauthor2Fragments[1][0])) **then**  19: //both authors have same middle full name or same first character of middle name  20: Count++  21: **endif**  22: **end elseif**  23: **elseif** (len(coauthor1Fragments) > 3 and len(coauthor2Fragments) > 3)) **then** //both have more than three name fragments  24: **if** ((coauthor1Fragments[0][0] == coauthor2Fragments[0][0] and coauthor1Fragments[0][1] == coauthor2Fragments[0][1] and coauthor1Fragments[0][2] == coauthor2Fragments[0][2]) and ((coauthor1Fragments[len(coauthor1Fragments)-1] == coauthor2Fragments[len(coauthor2Fragments)-1]) or (coauthor1Fragments[len(coauthor1Fragments)-1][0] == coauthor2Fragments[len(coauthor2Fragments)-1][0]))) **then**  //both have same first three characters of first name and either full last name or first character of last name  25: **if** (coauthor1Fragments[1][0] == coauthor2Fragments[1][0]) **then** //both have same first character of their second name  26: count++  27: **end if**  28: **end if**  29: **end elseif**  30: **end foreach**  31: **end foreach**  32: **if** (count ≥ 1) **then //**number ofsimilar co-authors excluding author in question  33: Flag ← true  34: **endif**  35: **return** Flag | | |
| **Algorithm 2 MHCF titleBasedClustering** | | **Algorithm 2.1 similarTitles** |
| **Input:** n groups Gn where each group has set of papers pr | | **Input: string literals of paper title, threshold\_value** |
| **Output:** Set of system generated clusters/groups **Gn** | | **Output: Boolean value** |
| 1: merge ← true, threshold-title ← threshold ← 0.85  2: **While** (merge ==’true’) **do:**  3: merge ← false  4: **for** i in range (0: len(G)-1):  5: **for** j in range (i+1: len(G)):  6: coathr-count = check-coauthors(Gi , Gj)  7: if coathr-count ≥ 1 **then**  8: threshold-title ← threshold – 0.35  9: **if** similarTitles(Gi\_title, Gj\_title, threshold-title) **then**  10: Gi ← Gi U Gj  11: G ← G.pop(j)  12: merge ← true  13: **end if**  14: **end for**  15: **end for**  16: **end while** | | 7.1: vectori, vectorj ← Research2Vec (titlei, titlej)  7.2: score = 1 – spatial.distance.cosine(vectori, vectorj)  7.3: **if** score ≥ threshold:  7.4: return true  7.5: **else**  7.6: return false  7.7: **end if** |
| **Algorithm 2.2 check-coauthors** | | |
| **Input:** Two groups co-authors list Gi and Gj  **Output:** Count of common co-authors (Number Value)  6.1: common = 0  6.2: if **set** (Gi) & **set** (Gj) **then** //if there exists a common co-author.  6.3: common ← **set** (Gi) & **set** (Gj)  6.4: return len(common) | | |
| **Algorithm 3 MHCF author\_affiliationBasedClustering** | **Algorithm 3.1 cosinesimilarity** | |
| **Input:** n groups Gn where each group has set of papers pr | **Input: List of affiliations/emails within the G, threshold** | |
| **Output:** Set of system generated clusters/groups Gn | **Output: Boolean value** | |
| 1: merge ← true  2: **While**(merge==’true’) **do:**  3: merge ← false  4: **for** i in range (0: len(G)-1):  5: **for** j in range (i+1: len(G)):  6: **if** cosinesimilarity (GiLaffiliation, GjLaffiliation) **then**  7: Gi ← Gi U Gj  8: G ← G.pop(j)  9: merge ← true  10: **end if**  11: **end for**  12: **end for**  13: **end while** | 6.1: **for** m in range (0: len(GiLaff) - 1):  6.2 vectori ← cosineSim (GiLaff[m])  6.3: **for** k in range (m+1: len(GjLaff)):  6.4: vectorj ← cosineSim (GjLaff[k])  6.5: score = 1 – spatial.distance.cosine(vectori, vectorj)  6.6: **if** score ≥ threshold:  6.7: return true  6.8: **end if**  6.9: **end for**  6.10: **end for**  6.11: return false | |

## External APIs/SDKs

Describe the third-party APIs/SDKs used in the project implementation in the following table. Few examples of APIs are provided in the table.

**Table 1 Details of APIs used in the project**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name of API and version** | **Description of API** | **Purpose of usage** | **List down the API endpoint/function/class in which it is used** |
| Stripe (version 2020-08-27) | Credit Card payment integration | Sandbox used for the orders payment | stripe.paymentMethods.create |
| Cloudinary | Image and Video management solution | Uploading Product Images on Cloudinary server | https://api.cloudinary.com/v1\_1/demo/image/upload |

## User Interface

Details about user interface with descriptions. Provide the User Interface for each sub-system (such as Mobile App, Web App, Client App, Admin App). Provide description of each User Interface explaining the details.

When inserting User Interfaces, use appropriate size of the image, for example, for mobile app, 2-4 screens can be placed on a single page.

Following are few examples of User Interfaces:

* + 1. **Login Screen**  **Home Screen**

Login screen of our mobile app where user Home screen where total, delayed and

have to choose its role and its company. Other complaints are shown.

A screenshot of a cell phone

Description automatically generated

Figure Home Screen

A screenshot of a cell phone

Description automatically generated

**Figure 2 Login Screen**

* + 1. **Assignee Dashboard**

Complain Assignee can view the graphs of month-wise complains, Resolved complains, summary and a list of submitted complains.

**A screenshot of a cell phone

Description automatically generated**

**Figure 3 Assignee Dashboard**

* + 1. **New Complaint**

A screenshot of a cell phone

Description automatically generatedComplain Assignee and Complainer can create a new complain by providing Description, Category, Title, Location etc.

Figure New Complaint

## Deployment

Specify the deployment environments used for hosting and live testing of all the sub-systems of the project. Provide the details of hosting/cloud service used, set of available software and their versions used etc.

# Testing and Evaluation



Once the system has been successfully developed, testing has to be performed to ensure that the system working as intended. This is also to check that the system meets the requirements stated earlier. Besides that, system testing will help in finding the errors that may be hidden from the user. The testing must be completed before it is deployed for use.

There are few types of testing which includes the unit testing, functional testing and integration testing.

You are *required* to perform each of these in-depth to ensure system quality.

## Unit Testing

It’s a level of software testing where individual units of a software/component are tested. The purpose is to validate that each unit of the software performs as designed.

**Unit Testing 1:** Login as Patient with valid and invalid credentials

**Testing Objective:** To ensure the login form is working correctly with valid and invalid credentials/inputs.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **No.** | **Test case/Test script** | **Attribute and value** | **Expected result** | **Result** |
| 1 | Check the email field of login to validate that it takes proper email | Email: [abc@gmail.com](mailto:abc@gmail.com) | Validates email address and moves cursor to next textbox | Pass |
| 2 | Check the email field of login to validate that it displays error message. | Email: [abc.gmail.com](mailto:abc@gmail.com) | Highlights field and displays error message | Pass |

## Functional Testing

The functional testing will take place after the unit testing. In this functional testing, the functionality of each of the module is tested. This is to ensure that the system produced meets the specifications and requirements.

**Functional Testing 1:** Login with different roles (Management, Patient, Doctor)

**Objective**: To ensure that the correct page with the correct navigation bar is loaded.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **No.** | **Test case/Test script** | **Attribute and value** | **Expected result** | **Actual result** | **Result** |
| 1. | Login as a ‘Management’ member. | Username: (correct username M003)  Password:  (correct password 1234) | Main page for the Management is loaded with the Management navigation bar. | Logged in and redirected to management main page. | Pass |
| 2. | Login as a ‘Doctor’ member. | Username: D003  Password:  1234 | Main page for the Doctor is loaded with the doctor navigation bar. | Login failed – invalid credentials error | Fail |

## Business Rules Testing

Decision table based testing technique is used to test business rules. The business rules were defined in FRs and Use Cases

Decision based testing uses a systematic approach where input and outputs are provided in tabular form. It is a precise and compact way to model complicated logic. The table contains conditions and actions are used for test cases where conditions as inputs and actions as outputs.

Detailed example is as given in Appendix E.

## Integration Testing

Integration tests assess whether a set of classes that must work together do so without error. They

ensure that the interfaces and linkages between different parts of the system work properly. At this point, the classes have passed their individual unit tests, so the focus now is on the flow of control among the classes and on the data exchanged among them. Integration testing follows the same general procedures as unit testing: The tester develops a test plan that has a series of tests, which, in turn, have a test. Integration testing is often done by a set of programmers and/or systems analysts.

**Integration Testing 1:** Scheduling Patient Appointment

**Testing Objective:** To ensure the scheduling is being done correctly and *the* ***interface*** *between* module ‘Patient/Doctor Management’ and module ‘Appointment/Scheduling’ *is running correctly*.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **No.** | **Test case/Test script** | **Attribute and value** | **Expected result** | **Actual result** | **Result** |
| 1. | Make Appointment | Doctor schedule, patient preferred date and time | Successfully create doctor-patient appointment showing date and time of appointment. | Appointment created successfully | Pass |
| 2. | Change Appointment booking | Select Date and time | Final date and time of appointment will be shown. | Appointment time changed successfully | Pass |

# Appendix A

**Box-and-line diagram**

Box-and-line diagrams are often used to describe the business concepts and processes during the analysis phase of the software development lifecycle. These diagrams come with descriptions of components and connectors, as well as other descriptions that provide common inherent interpretations.

**Example:**

Lines in the box-and-line diagrams indicate the relationship among components

Figure A-1 Box-and-Line Diagram for an Online Shopping Business

* The semantic of lines may refer dependency, control flow, data flow, etc
* Lines may be associated with arrows to indicate the process direction and sequence.
* A box-and-line diagram can be used as a business concept diagram describing its application domain and process concepts

**Example of Architecture Pattern:**

The **figure A-2** shows an example of the logical package organization of the layered architecture. The top level deals with user interface, the next level is for utilities, and the one below utility provides core services. Each layer gets support from its lower adjacent layer by an interface implementation and from the related classes in the same layer.

A simple software system may consist of two layers: an interaction layer and a processing layer:

* The interaction layer provides user interfaces to clients, takes requests, validates and forwards requests to the processing layer for processing, and responds to clients.
* The processing layer receives the forwarded requests and performs the business logic process, accesses the database, returns the results to its upper layer, and lets the upper layer respond to clients since the upper layer has the GUI interface responsibility.

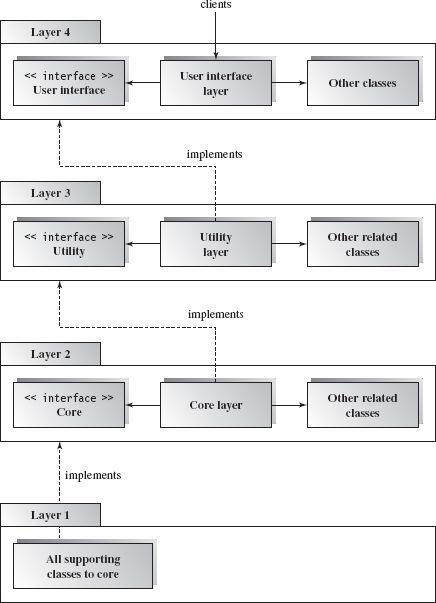


Figure A-2 Component-based Layered Architecture

**Note:** The Architecture pattern shall be selected according to the targeted system’s requirements and quality attributes. Above example is provided to demonstrate that how the system architecture is required to be presented.

# Appendix B

**Design Models**

**Activity Diagram**

Following activity diagram is of an appointment system presenting **make an appointment** process in which all diagram’s elements are presented. In further in **Table B-1** to the detail of activity diagram syntax is provided.

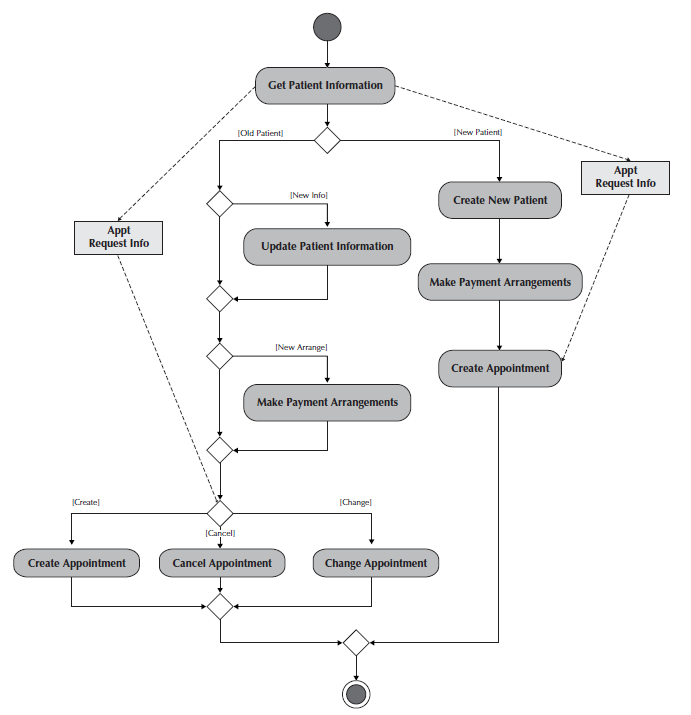
**Example**

Figure B-1 Activity Diagram for Make an Appointment Process

**Activity Diagram Syntax**

Table B- 1Activity Diagram Syntax

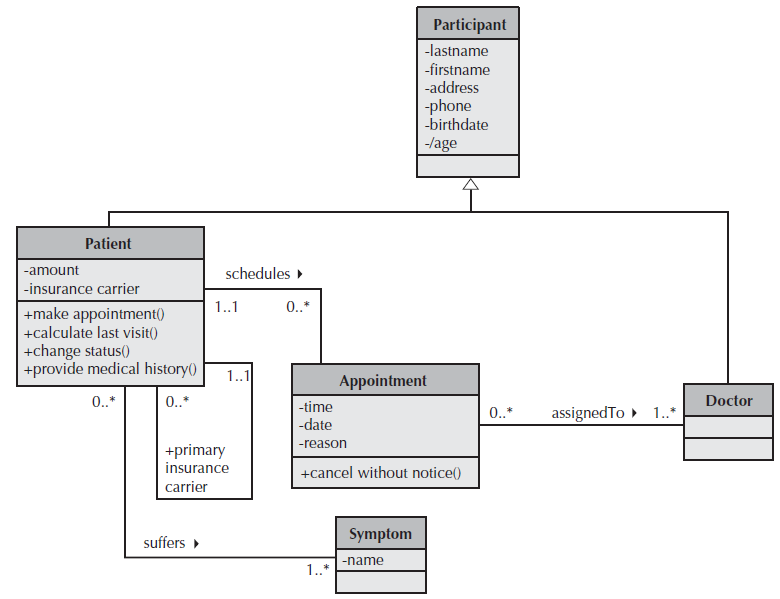
|  |  |
| --- | --- |
| **Term and definition** | **Symbol** |
| **An action:**   * Is a simple, non-decomposable piece of behavior. * Is labeled by its name. |  |
| **An activity:**   * Is used to represent a set of actions. * Is labeled by its name. |  |
| **An object node:**   * Is used to represent an object that is connected to a set of object flows. * Is labeled by its class name. |  |
| **A control flow:**   * Shows the sequence of execution. |  |
| **An object flow:**   * Shows the flow of an object from one activity (or action) to another activity (or action). |  |
| An initial node:   * Portrays the beginning of a set of actions or activities. |  |
| **A final-activity node:**   * Is used to stop all control flows and object flows in an activity (or action). |  |
| **A final-flow node:**   * Is used to stop a specific control flow or object flow. |  |
| **A decision node:**   * Is used to represent a test condition to ensure that the control flow or object flow only goes down one path. * Is labeled with the decision criteria to continue down the specific path. |  |
| **A merge node:**   * Is used to bring back together different decision paths that were created using a decision node. |  |
| **A fork node:**   * Is used to split behavior into a set of parallel or concurrent flows of activities (or action) |  |
| **A join node:**   * Is used to bring back together a set of parallel or concurrent flows of activities (or action) |  |
| **A swimlane:**   * Is used to break up an activity diagram into rows and columns to assign the individual activities (or actions) to the individuals or objects that are responsible for executing the activity (or action) * Is labeled with the name of the individual or object responsible |  |

**Class Diagram**

Following class diagram is of an appointment system in which all class diagrams elements are presented. In further in **Table B-2** to the detail of class diagram syntax is provided.

**Example**

Figure B-2 Class Diagram for an Appointment System



**Class Diagram Syntax**

Table B- 2 Class Diagram Syntax

|  |  |
| --- | --- |
| **Term and definition** | **Symbol** |
| **A class:**   * Has a name typed in bold and centered in its top compartment. * Has a list of attributes in its middle compartment. * Represents a kind of person, place, or thing about which the system will need to capture and store information. * Has a list of operations in its bottom compartment. * Does not explicitly show operations that are available to all classes. |  |
| **An attribute:**   * Represents properties that describe the state of an object. * Can be derived from other attributes, shown by placing a slash before the attribute’s name. | attribute name  /derived attribute name |
| **An operation:**   * Represents the actions or functions that a class can perform. * Can be classified as a constructor, query, or update operation. * Includes parentheses that may contain parameters or information needed to perform the operation. | operation name () |
| **An association:**   * Represents a relationship between multiple classes or a class and itself. * Is labeled using a verb phrase or a role name, whichever better represents the relationship. * Can exist between one or more classes. * Contains multiplicity symbols, which represent the minimum and maximum times a class instance can be associated with the related class instance. |  |
| **A generalization:**   * Represents a-kind-of relationship between multiple classes. |  |
| **An aggregation:**   * Represents a logical a-part-of relationship between multiple classes or a class and itself. * Is a special form of an association. |  |
| **A composition:**   * Represents a physical a-part-of relationship between multiple classes or a class and itself * Is a special form of an association. |  |

**Sequence Diagram**

Following example shows an instance sequence diagram that depicts the objects and messages for the Make Old Patient Appt use case, which describes the process by which an existing patient creates a new appointment or cancels or reschedules an appointment. In further in **Table B-3** to the detail of class diagram syntax is provided.

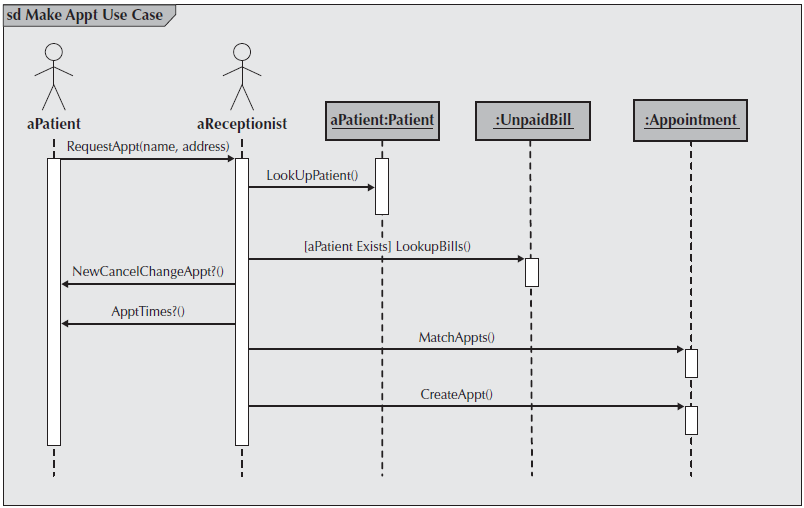
**Example**

Figure B-3 Example Sequence Diagram

**Sequence diagram Syntax**

Table B-3 Sequence Diagram Syntax

|  |  |
| --- | --- |
| **Term and definition** | **Symbol** |
| **An actor:**   * Is a person or system that derives benefit from and is external to the system. * Participates in a sequence by sending and/or receiving messages. * Is placed across the top of the diagram. * Is depicted either as a stick figure (default) or, if a nonhuman actor is involved, as a rectangle with <<actor>> in it (alternative). |  |
| **An object:**   * Participates in a sequence by sending and/or receiving messages. * Is placed across the top of the diagram. |  |
| **A lifeline:**   * Denotes the life of an object during a sequence. * Contains an X at the point at which the class no longer interacts. |  |
| **An execution occurrence:**   * Is a long narrow rectangle placed atop a lifeline. * Denotes when an object is sending or receiving messages. |  |
| **A message:**   * Conveys information from one object to another one. * An operation call is labeled with the message being sent and a solid arrow, whereas a return is labeled with the value being returned and shown as a dashed arrow. |  |
| **A guard condition:**   * Represents a test that must be met for the message to be sent. |  |
| **For object destruction:**   * An X is placed at the end of an object’s lifeline to show that it is going out of existence. |  |
| **A frame:**   * Indicates the context of the sequence diagram. |  |

**Behavioral State Machine Diagram**

Following example of a behavioral state machine representing the patient class in the context of a hospital environment. From this diagram, we can tell that a patient enters a hospital and is admitted after checking in. If a doctor finds the patient to be healthy, he or she is released and is no longer considered a patient after two weeks elapse. If a patient is found to be unhealthy, he or she remains under observation until the diagnosis changes.

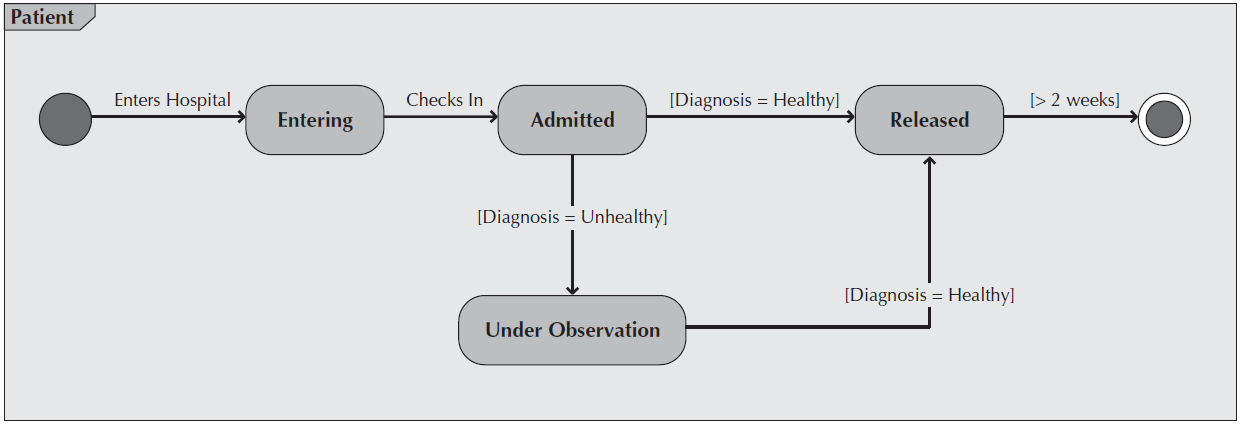
**Example**

Figure B-4 Sample Behavioral State Machine Diagram

**Behavioral State Machine Diagram Syntax**

Table B-4 Behavioral State Machine Diagram Syntax

|  |  |
| --- | --- |
| **Term and definition** | **Symbol** |
| **A state:**   * Is shown as a rectangle with rounded corners. * Has a name that represents the state of an object. |  |
| **An initial state:**   * Is shown as a small, filled-in circle. * Represents the point at which an object begins to exist. |  |
| **A final state:**   * Is shown as a circle surrounding a small, filled-in circle (bull's-eye). * Represents the completion of activity. |  |
| **An event:**   * Is a noteworthy occurrence that triggers a change in state. * Can be a designated condition becoming true, the receipt of an explicit signal from one object to another, or the passage of a designated period of time. * Is used to label a transition. |  |
| **A transition:**   * Indicates that an object in the first state will enter the second state. * Is triggered by the occurrence of the event labeling the transition. * Is shown as a solid arrow from one state to another, labeled by the event name. |  |
| **A frame:**   * Indicates the context of the behavioral state machine. |  |

**Data Flow Diagram**

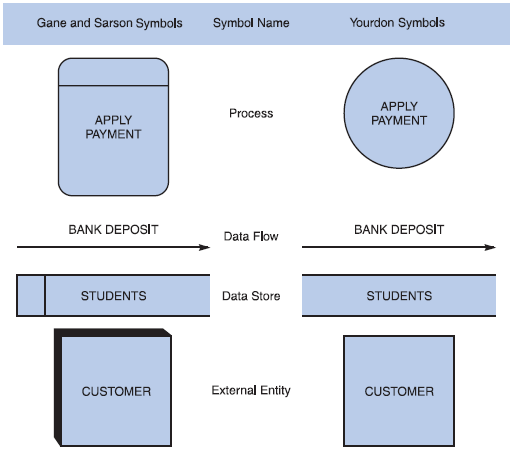
**Data flow diagram symbols, symbol names, and examples**

Figure B-5 Data flow diagram symbols, symbol names, and examples of the Gane and Sarson and Yourdon symbol sets.

**Guidelines for Drawing DFDs**

**Step 1: Draw a Context Diagram**: The first step in constructing a set of DFDs is to draw a context diagram. A **context diagram** is a top-level view of an information system that shows the system’s boundaries and scope. Data stores are not shown in the context diagram because they are contained within the system and remain hidden until more detailed diagrams are created.

**Example**

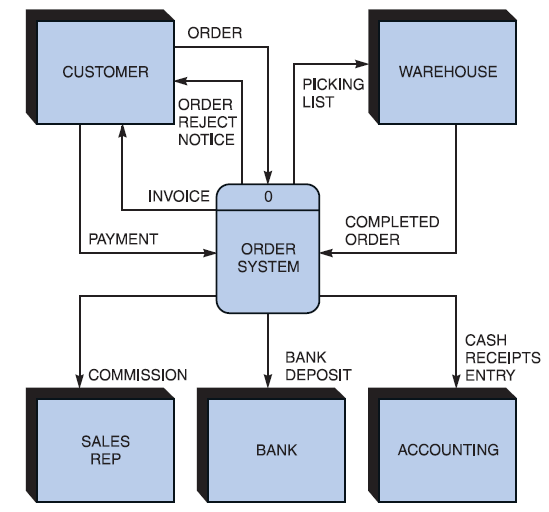


Figure B-6 Context diagram DFD for an order system.

**Step 2: Draw a Diagram 0 DFD:** To show the detail inside the black box, you create DFD diagram 0. **Diagram 0** zooms in on the system and shows major internal processes, data flows, and data stores. Diagram 0 also repeats the entities and data flows that appear in the context diagram. When you expand the context diagram into DFD diagram 0, you must retain all the connections that flow into and out of process 0.

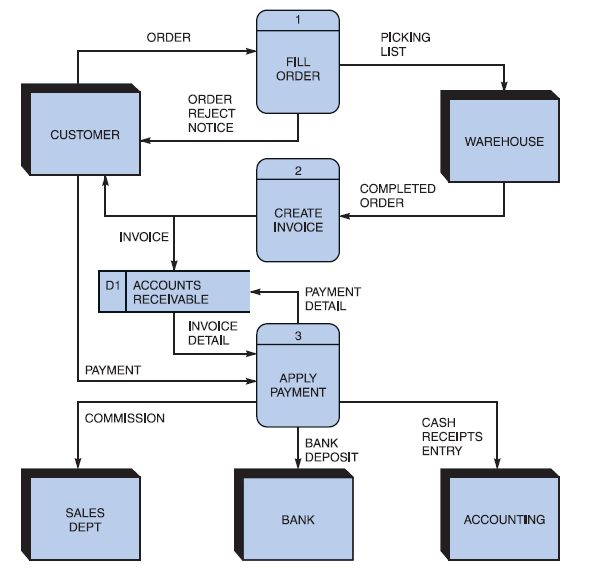
**Example**

Figure B-7 Diagram 0 DFD for the order system.

**Step 3: Draw the Lower-Level Diagrams:**

To create lower-level diagrams, you must use leveling and balancing techniques. **Leveling** is the process of drawing a series of increasingly detailed diagrams, until all functional primitives are identified.

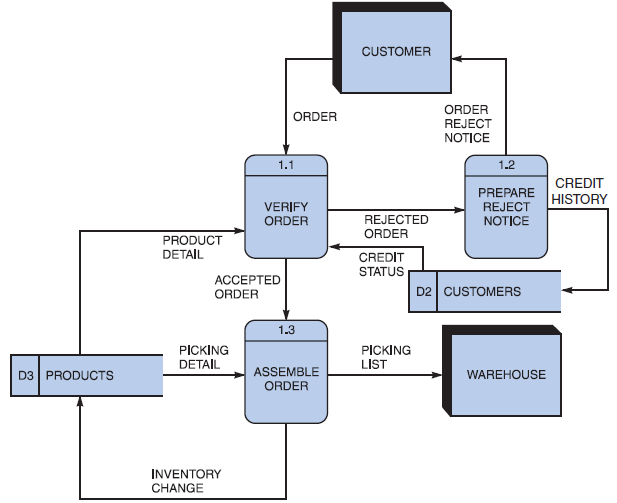
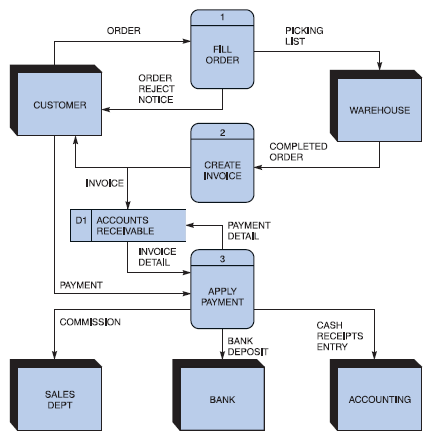
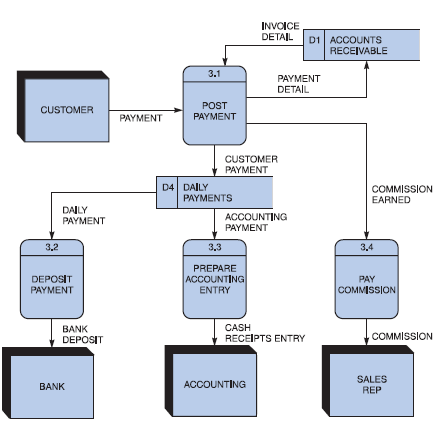
**Leveling Example**

Figure B-8 Diagram 1 DFD shows details of the FILL ORDER process in the order system.

**Balancing** maintains consistency among a set of DFDs by ensuring that input and output data flows align properly.

**Balancing Example**

Order System Diagram 0 DFD

Order System Diagram 3 DFD

The order system diagram 0 is shown at the top of the figure and exploded diagram 3 DFD (for the APPLY PAYMENT process) is shown at the bottom. The two DFDs are balanced, because child diagram at the bottom has the same input and output flows as the parent process 3 shown at the top.

## Appendix D: General Coding Standards & Guidelines

1. Follow a consistent variable naming convention throughout the code. E.g.
   * Snakecase (words are delimited by “\_” like: variable\_one)
   * Pascalcase (words are delimited by capital letters like: VariableOne)
   * Camelcase (words are delimited by capital letters except the initial word like: variableOne)
   * Hungarian Notation (describes the variable type or purpose at the start of the variable name like: arrDistributeGroup)
2. Use naming that visually describes scope like privateField, Const etc
3. Use read only/immutable when a field’s value should not be changed after initialization
4. Use only get, for properties that should not be updated from outside
5. Name functions according to their functionalities.
6. Insert appropriate comments to make the code understandable to any reader. Additionally follow a consistent style to do so. E.g.

/\* the below function will be used for the addition of two variables\*/

int Add(){

//logic of the function

}

Avoid commenting on obvious things

1. Make use of indentation for indicating the start and end of the control structures along with a clear specification of where the code is between them.
2. Follow consistent naming convention for files and folders.
3. Follow proper structure for classes
4. Group code entities logically into projects/packages/modules/folders
   1. Separate logical layers of application into different modules/services/utilities etc.
   2. User separate files for each class, struct, interface, enum etc. Name of the file and the enclosing entity must be same. E.g., class Employee in Employee.cs/Employee.java
5. Define and use everything within the minimum scope possible
6. Use proper access modifier for all code entities if required
7. Code entities should have maximum cohesion and least coupling possible.
8. Follow DRY law.
   1. Do not repeat code.
   2. A piece of knowledge should exist only in one place within the codebase/application
   3. Reuse code as much as possible
   4. Always write short methods
   5. Single method should not have too many logic, long conditional flow or too many parameters
9. Strictly follow Single Responsibility Principle (SRP) when writing methods, classes, modules, projects, packages, or any other code entities.
10. Write classes and other code entities that are easy to extend without modification.
11. Handle exceptions
12. Log exception and other significant event details
13. Follow a consistent convention for logging all over the application

**Appendix E: Business rules testing**

Methodology for creating decision table:

Table

Description automatically generated

**Example:**

The provided example is of a super store.

Table

Description automatically generated

**Table

Description automatically generated**

Table

Description automatically generated

Table, calendar

Description automatically generated

Table

Description automatically generated

* Now Combine rules where it is apparent that an alternative does not make a difference in the outcome

Calendar

Description automatically generated

This is the final table and now you have to create test cases on every rule. In above example there are 6 rules so there shall be 6 test cases.