CHAPTER II

LITERATURE REVIEW
CHAPTER II

LITERATURE REVIEW

2.1 C# Programming

Visual C# or commonly used by C Sharp is a programming language used in various aspects with general purpose. This Programming Language is commonly used to create server-side on websites, create desktop applications as needed, mobile applications, programming games, sales applications, absences and many more which are all general.

Microsoft Visual C# is a program of programming tools (Rapid Application Development Tool) created by Microsoft Corporation companies and can be used to create graphics-based programs using programming languages that are almost similar in pattern, namely: C ++ Programming.[2]

2.4.1 History of C#

When the beginning of the Windows operating system was introduced by Microsoft Corp. to replace the DOS operating system (Disk Operating System), programmers generally use the C programming language to create application programs on it. The reason is because Windows (Application Programming Interface) APIs that are widely used by application programs are also written using C language. This can only be done well by talented programmers. For most others, writing programs using C language feels very difficult because when writing an application program using C language, programming must manually allocate memory. In addition, the complexity of the C programming language is also complicated by the presence of arithmetic pointers which are also complex and the program structure is relatively difficult to read.

After the birth of the C ++ language created by Bjarne Stroustrup which was developed with the concept of object-oriented programming language (OOP), many C programmers switched to C ++ to do programming on Windows operating systems. In this case the C ++ language can be considered as an extension of the C language. It turns out that the C ++ programming language is even as complex as the C language, the C ++ language is also the same as C language which manually allocates memory with the help of a pointer on C ++ language syntax. Although very complicated, some C ++ frameworks are still present today, for example MFC (Microsoft Foundation Classes) which provides classes that can be used to develop Win32 API applications (the classes used to develop applications on Windows operating systems that have 32 bit mode).[3]
2.4.2 IDEs For C#

An integrated development environment (IDE) is a software suite that consolidates the basic tools developers need to write and test software. Typically, an IDE contains a code editor, a compiler or interpreter and a debugger that the developer accesses through a single graphical user interface (GUI). An IDE may be a standalone application, or it may be included as part of one or more existing and compatible applications.

Open source IDEs in C#

1. Sharp Develop

Sharp Develop (short for Sharp Develop) is a free IDE for C# and VB.NET projects on Microsoft’s .NET platform.

2. C# Studio

C# Studio is a simple IDE for a C#/Mono/GTK# developer.

3. Mono Develop

Mono Develop is a project to port Sharp Develop to GTK#. There are numerous goals that Mono Develop hopes to achieve. Some of these are: To create a best of breed development environment for Unix systems for C# and Mono. Since it written in GTK#, and we like GTK# and we get good support from GTK#, most likely it will add functionality to improve the GTK# experience. Today the IDE is a simple IDE and on Unix does not do GUI design, but we plan on adding a GUI designer. We want to integrate the tools we have been building so far, so things like Mono Doc, NUnit-Gtk and the debugger should target MonoDevelop.

4. Fireball.Code Editor

Fireball.Code Editor is a source editor control with syntax highlight support. It supports some common programming language and you can add your own syntax. Also on the website you can find a software called FireEdit. It is a open source small code editor with support for extensibility from plugins system, more info on the web site, join the forum and help the staff to add feature and find bugs, by testing the control or the application or by making a plugin.

5. Xacc

Xacc.ide is a small and fast integrated development environment (IDE) mainly targeted at .NET development. It has syntax support for 27 languages and project support for various compilers. It has support for step debugging .NET based executables. Also featured is brace matching and folding support for some languages.
2.2 Microsoft Visual Studio

Visual Studio .NET is a Microsoft-integrated development environment (IDE) that can be used for developing consoles, graphical user interfaces (GUIs), Windows Forms, Web services and Web applications.

Visual Studio is used to write native code and managed code supported by Microsoft Windows, Windows Mobile, Windows CE, .NET Framework, .NET Compact Framework and Microsoft Silverlight. Visual Studio .NET’s code editor supports IntelliSense and code refactoring, while the Visual Studio .NET integrated debugger supports both source and machine-level debugging. Visual Studio .NET includes other built-in tools, like a form designer, which is useful when building GUI applications; a Web designer that creates dynamic Web pages; a class designer that is used to create custom libraries, and a schema designer for database support.[5]

2.2.1 History of Visual Studio

Although this book is dedicated to Visual Studio 2010 and .NET Framework 4.0, having a good historical background in Visual Studio can help you better understand the features treated in the subsequent chapters. Regardless of whether you are old friends with Visual Studio or it is new for you, it is always worth knowing where it started and how it’s been evolving.

The roots of Visual Studio go back for almost 19 years, back to the point somewhere between the release of Windows 3.0 and 3.1. It is incredible how the development tool has evolved enormously during almost two decades! The road behind Visual Studio was never smooth or flat; it was full of bumps and curves. However, one thing stayed constant during the years: Microsoft created this tool with developers in mind, and made amazing efforts to build a strong developer community surrounding the product.

For a long time, Windows development was a field where only C and C++ programmers could play. They had to carry out a lot of tasks for creating the simplest user interface — such as defining and registering Windows classes, implementing the Windows message loop, dispatching Windows messages, painting the client in Windows, and so on. The smallest “Hello, World” program for Windows was about a hundred lines of code, where you could not meet any explicit statement to print out the “Hello, World” text. Instead, you had to draw this text to an absolute window position as a response to the WM_PAINT message. At that time, the user interface (UI) was defined by static text files that were compiled into binary resources and linked to the application. The UI missed the concept of controls there were windows and child windows, all of them represented by HWNDs (or window handles).[6]
2.3 SQL Server

SQL Server is a relational database management system, or RDBMS, that supports a wide variety of transaction processing, business intelligence and analytics applications in corporate IT environments. It's one of the three market-leading database technologies, along with Oracle Database and IBM's DB2.

Like other RDBMS software, Microsoft SQL Server is built on top of SQL, a standardized programming language that database administrators (DBAs) and other IT professionals use to manage databases and query the data they contain. SQL Server is tied to Transact-SQL (T-SQL), an implementation of SQL from Microsoft that adds a set of proprietary programming extensions to the standard language.

The original SQL Server code was developed in the 1980s by the former Sybase Inc., which is now owned by SAP. Sybase initially built the software to run on Unix systems and minicomputer platforms. It, Microsoft and Ashton-Tate Corp., then the leading vendor of PC databases, teamed up to produce the first version of what became Microsoft SQL Server, designed for the OS/2 operating system and released in 1989.

Ashton-Tate stepped away after that, but Microsoft and Sybase continued their partnership until 1994, when Microsoft took over all development and marketing of SQL Server for its own operating systems. The year before, with the Sybase relationship starting to unravel, Microsoft had also made the software available on the newly released Windows NT after modifying the 16-bit OS/2 code base to create a 32-bit implementation with added features; it focused on the Windows code going forward. In 1996, Sybase renamed its version Adaptive Server Enterprise, leaving the SQL Server name to Microsoft[7].

2.4 UML (Unified Modeling Language)

UML, short for Unified Modeling Language, is a standardized modeling language consisting of an integrated set of diagrams, developed to help system and software developers for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems. The UML is a very important part of developing object oriented software and the software development process. The UML uses mostly graphical notations to express the design of software projects. Using the UML helps project teams communicate, explore potential designs, and validate the architectural design of the software. In this article we will give you detailed ideas about what
is UML, the history of UML and a description of each UML diagram type, along with UML examples.

2.4.1 The Origin of UML

The goal of UML is to provide a standard notation that can be used by all object-oriented methods and to select and integrate the best elements of precursor notations. UML has been designed for a broad range of applications. Hence, it provides constructs for a broad range of systems and activities (e.g., distributed systems, analysis, system design and deployment).

UML is a notation that resulted from the unification of OMT from.

2. Booch [Grady Booch 1994] - was excellent for design and implementation. Grady Booch had worked extensively with the Ada language, and had been a major player in the development of Object Oriented techniques for the language. Although the Booch method was strong, the notation was less well received (lots of cloud shapes dominated his models - not very tidy)
3. OOSE (Object-Oriented Software Engineering [Ivar Jacobson 1992]) - featured a model known as Use Cases. Use Cases are a powerful technique for understanding the behavior of an entire system (an area where OO has traditionally been weak).

In 1994, Jim Rumbaugh, the creator of OMT, stunned the software world when he left General Electric and joined Grady Booch at Rational Corp. The aim of the partnership was to merge their ideas into a single, unified method (the working title for the method was indeed the "Unified Method").

By 1995, the creator of OOSE, Ivar Jacobson, had also joined Rational, and his ideas (particularly the concept of "Use Cases") were fed into the new Unified Method - now called the Unified Modelling Language. The team of Rumbaugh, Booch and Jacobson are affectionately known as the "Three Amigos"

UML has also been influenced by other object-oriented notations:

- Mellor and Shlaer [1998]
- Coad and Yourdon [1995]
- Wirfs-Brock [1990]
- Martin and Odell [1992]

UML also includes new concepts that were not present in other major methods at the time, such as extension mechanisms and a constraint language.
2.4.2 History of UML

1. During 1996, the first Request for Proposal (RFP) issued by the Object Management Group (OMG) provided the catalyst for these organizations to join forces around producing a joint RFP response.

2. Rational established the UML Partners consortium with several organizations willing to dedicate resources to work toward a strong UML 1.0 definition. Those contributing most to the UML 1.0 definition included:
   - Digital Equipment Corp
   - HP
   - i-Logix
   - IntelliCorp
   - IBM
   - ICON Computing
   - MCI Systemhouse
   - Microsoft
   - Oracle
   - Rational Software
   - TI
   - Unisys

3. This collaboration produced UML 1.0, a modeling language that was well-defined, expressive, powerful, and generally applicable. This was submitted to the OMG in January 1997 as an initial RFP response.1

4. In January 1997 IBM, ObjecTime, Platinum Technology, Ptech, Taskon, Reich Technologies and Softeam also submitted separate RFP responses to the OMG. These companies joined the UML partners to contribute their ideas, and together the partners produced the revised UML 1.1 response. The focus of the UML 1.1 release was to improve the clarity of the UML 1.0 semantics and to incorporate contributions from the new partners. It was submitted to the OMG for their consideration and adopted in the fall of 1997.1 and enhanced 1.1 to 1.5, and subsequently to UML 2.1 from 01 to 06 (now the UML current version is 2.5)[8].
2.5 Use Case Diagram

Use case diagram is the primary form of system/software requirements for a new software program under developed. Use cases specify the expected behaviour (what), and not the exact method of making it happen (how). Use cases once specified can be denoted both textual and visual representation (such as UML). A key concept of use case modeling is that it helps us design a system from end user's perspective. It is an effective technique for communicating system behavior in the user's terms by specifying all externally visible system behavior.[9]

Basic use case diagram symbols and notation

System
Draw your system's boundaries using a rectangle that contains use cases. Place actors outside the system's boundaries.

![System Diagram](image1)

Use Case
Draw use cases using ovals. Label the ovals with verbs that represent the system's functions.

![Use Case Diagram](image2)

Actors
Actors are the users of a system. When one system is the actor of another system, label the actor system with the actor stereotype.

![Actor Diagram](image3)
Relationships

Illustrate relationships between an actor and a use case with a simple line. For relationships among use cases, use arrows label either "uses" or "extends." A "uses" relationship indicates that one use case is needed by another in order to perform a task. An "extends" relationship indicates alternative options under a certain use case.

2.6 Context Diagram

The Context Diagram shows the system under consideration as a single high-level process and then shows the relationship that the system has with other external entities (systems, organizational groups, external data stores, etc.). Another name for a Context Diagram is a Context-Level Data-Flow Diagram or a Level-0 Data Flow Diagram.

A DFD is process centric and depicts 4 main components.

- Processes (circle)
- External Entities (rectangle)
- Data Stores (two horizontal, parallel lines or sometimes and ellipse)
- Data Flows (curved or straight line with arrowhead indicating flow direction)

Each DFD may show a number of processes with data flowing into and out of each process. If there is a need to show more detail within a particular process, the process is decomposed into
a number of smaller processes in a lower level DFD. In this way, the Context Diagram or Context-Level DFD is labeled a “Level-0 DFD” while the next level of decomposition is labeled a “Level-1 DFD”, the next is labeled a “Level-2 DFD”, and so on.

Context Diagrams and Data-Flow Diagrams were created for systems analysis and design. But like many analysis tools they have been leveraged for other purposes. For example, they can also be leveraged to capture and communicate the interactions and flow of data between business processes. So, they don’t have to be restricted to systems analysis.

A Context Diagram (and a DFD for that matter) provides no information about the timing, sequencing, or synchronization of processes such as which processes occur in sequence or in parallel. Therefore it should not be confused with a flowchart or process flow which can show these things.

Some of the benefits of a Context Diagram are:

- Shows the scope and boundaries of a system at a glance including the other systems that interface with it
- No technical knowledge is assumed or required to understand the diagram
- Easy to draw and amend due to its limited notation
- Easy to expand by adding different levels of DFDs
- Can benefit a wide audience including stakeholders, business analyst, data analysts, developers

A number of factors should be considered when deciding what approach to use (top down or bottom up) to create a Context Diagram. First let’s be clear about what each approach means. The top-down approach: this approach refers to starting directly with the Context Diagram and creating it from scratch. This is done by placing the system under consideration in the center and then identifying each of its interactions based on either the analyst’s current knowledge or by asking others who may have the knowledge.

The bottom-up approach: this approach refers to starting with a lower level data-flow diagram, identifying known processes and the data that flows between them, and then following the data trail to uncover other unknown processes. After the lower-level data-flow diagram is completed, the processes can be rolled up to the system level.

The top-down approach is a more direct approach and is often the direction many analysts choose to take (whether right or wrong for their particular situation). It can work well under some
circumstances, particularly if the system under consideration has only a few interactions with external entities or if the same analysts support the application on an ongoing basis and are very familiar with all of the possible interactions and dependencies that the system has. But if the system has a very large number of interactions and dependencies or if the analyst is unfamiliar with the system then this approach can be difficult. The less informed analyst would need to rely on a very iterative approach where the context diagram is evolved over time by reviewing it with a number of subject matter experts that understand pieces of the environment.

The bottom-up approach is a less direct method, but can be exceptionally valuable as an investigative technique. Starting with a lower-level data flow diagram allows the analyst to document the bits and pieces of the system that are understood and then follow the data to identify the various system interactions and dependencies that exist. This tends to be a more natural approach as it uses the power of the technique to document the entire scope of the system and its surrounding environment. In the top-down approach it can be difficult to know when the analyst has reached completion since it’s hard to know that which you don’t know. In the bottom-up approach the data-flow diagramming technique itself drives the analyst to a state of completion.\[11\]

### 2.7 Flow Chart Diagram

The flowchart is a means to visually present the flow of data through an information processing systems, the operations performed within the system and the sequence in which they are performed. In this lesson, we shall concern ourselves with the program flowchart, which describes what operations (and in what sequence) are required to solve a given problem. The program flowchart can be likened to the blueprint of a building. As we know, a designer draws a blueprint before starting to construct a building. Similarly, a programmer prefers to draw a flowchart prior to writing a computer program. As in the case of the drawing of a blueprint, the flowchart is drawn according to defined rules and using standard flowchart symbols prescribed by the American National Standard Institute, Inc. A flowchart is a diagrammatic representation that illustrates the sequence of operations to be performed to get the solution of a problem. Flowcharts are generally drawn in the early stages of formulating computer solutions. Flowcharts facilitate communication between programmers and business people. These flowcharts play a vital role in the programming of a problem and are quite helpful in understanding the logic of complicated and lengthy problems. Once the flowchart is drawn, it becomes easy to write the program in any high level language. Often we see how flowcharts are helpful in explaining the program to others. Hence, it is correct to say that a flowchart is a must for the better documentation of a complex program.
Flow is a representation of a series of logic operations to satisfy specific requirements. A flow exists naturally. It can be irregular, unfixed or full of problems. For this reason, it may apparently be absent in some situations. Lately, members of a team were assigned to investigate the flow of a business process, and I was told that there were some deficiencies in the flow. The reply from the person who was in charge of the team was that no flow was shown in part of the business process. As a matter of fact, it is impossible for a business carried out without a flow. It may be a flow in an unfixed form, or, may be the person himself whom you investigated does not have a clear sense about the flow.

Chart, or diagram, is a presentation or a written description of some regular and common parts of the flow. A chart is conducive to communication and concentration and offers references for process reengineering.

Flow chart can be seen from the definition that a flow accompanies always with business or transaction. Not all of the flows, however, are appropriate to be expressed by flowcharts. Flows that can be expressed by charts follow some fixed routines, and the key links of flows won't be changed constantly.

A flowchart helps to clarify how things are currently working and how they could be improved. It also assists in finding the key elements of a process, while drawing clear lines between where one process ends and the next one starts. Developing a flowchart stimulates communication among participants and establishes a common understanding about the process. Flowcharts also uncover steps that are redundant or misplaced. In addition, flowcharts are used to identify appropriate team members, to identify who provides inputs or resources to whom, to establish important areas for monitoring or data collection, to identify areas for improvement or increased efficiency, and to generate hypotheses about causes. Flowcharts can be used to examine processes for the flow of patients, information, materials, clinical care, or combinations of these processes. It is recommended that flowcharts be created through group discussion, as individuals rarely know the entire process and the communication contributes to improvement.

Types of flowcharts:

High-Level Flowchart

A high-level (also called first-level or top-down) flowchart shows the major steps in a process. It illustrates a "birds-eye view" of a process, such as the example in the figure entitled High-Level Flowchart of Prenatal Care. It can also include the intermediate outputs of each step (the product or service produced), and the sub-steps involved. Such a flowchart offers a basic
picture of the process and identifies the changes taking place within the process. It is significantly useful for identifying appropriate team members (those who are involved in the process) and for developing indicators for monitoring the process because of its focus on intermediate outputs. Most processes can be adequately portrayed in four or five boxes that represent the major steps or activities of the process. In fact, it is a good idea to use only a few boxes, because doing so forces one to consider the most important steps. Other steps are usually sub-steps of the more important ones.

**Detailed Flowchart**

The detailed flowchart provides a detailed picture of a process by mapping all of the steps and activities that occur in the process. This type of flowchart indicates the steps or activities of a process and includes such things as decision points, waiting periods, tasks that frequently must be redone (rework), and feedback loops. This type of flowchart is useful for examining areas of the process in detail and for looking for problems or areas of inefficiency. For example, the Detailed Flowchart of Patient Registration reveals the delays that result when the record clerk and clinical officer are not available to assist clients.

**Deployment or Matrix Flowchart**

A deployment flowchart maps out the process in terms of who is doing the steps. It is in the form of a matrix, showing the various participants and the flow of steps among these participants. It is chiefly useful in identifying who is providing inputs or services to whom, as well as areas where different people may be needlessly doing the same task. See the Deployment of Matrix Flowchart.

Each type of flowchart has its strengths and weaknesses; the high-level flowchart is the easiest to construct but may not provide sufficient detail for some purposes. In choosing which type to use, the group should be clear on their purpose for flowcharting. The table below entitled "Type of Flowchart Indicated for Various Purposes" gives some indications, but if you're unsure which to use, start with the high-level one and move on to detailed and deployment. Note that the detailed and deployment flowcharts are time-consuming.

**A Set of Useful Standard Flowchart Symbols**

It is not strictly necessary to use boxes, circles, diamonds or other such symbols to construct a flowchart, but these do help to describe the types of events in the chart more clearly. Described
below are a set of standard symbols which are applicable to most situations without being overly complex.

- Rounded box - use it to represent an event which occurs automatically. Such an event will trigger a subsequent action, for example `receive telephone call', or describe a new state of affairs.
- Rectangle or box - use it to represent an event which is controlled within the process. Typically this will be a step or action which is taken. In most flowcharts this will be the most frequently used symbol.
- Diamond - use it to represent a decision point in the process. Typically, the statement in the symbol will require a `yes' or `no' response and branch to different parts of the flowchart accordingly.
- Circle - use it to represent a point at which the flowchart connects with another process. The name or reference for the other process should appear within the symbol.[12]

![Flow Chart Symbol](image)

Figure 2.6 Flow Chart Symbol.
2.8 DFD (Data Flow Diagram)

A data flow diagram (DFD) maps out the flow of information for any process or system. It uses defined symbols like rectangles, circles and arrows, plus short text labels, to show data inputs, outputs, storage points and the routes between each destination. Data flowcharts can range from simple, even hand-drawn process overviews, to in-depth, multi-level DFDs that dig progressively deeper into how the data is handled. They can be used to analyze an existing system or model a new one. Like all the best diagrams and charts, a DFD can often visually “say” things that would be hard to explain in words, and they work for both technical and nontechnical audiences, from developer to CEO. That’s why DFDs remain so popular after all these years. While they work well for data flow software and systems, they are less applicable nowadays to visualizing interactive, real-time or database-oriented software or systems.

History of the DFD

Data flow diagrams were popularized in the late 1970s, arising from the book Structured Design, by computing pioneers Ed Yourdon and Larry Constantine. They based it on the “data flow graph” computation models by David Martin and Gerald Estrin. The structured design concept took off in the software engineering field, and the DFD method took off with it. It became more popular in business circles, as it was applied to business analysis, than in academic circles.

Also contributing were two related concepts:

- Object Oriented Analysis and Design (OOAD), put forth by Yourdon and Peter Coad to analyze and design an application or system.
- Structured Systems Analysis and Design Method (SSADM), a waterfall method to analyze and design information systems. This rigorous documentation approach contrasts with modern agile approaches such as Scrum and Dynamic Systems Development Method (DSDM.)

Three other experts contributing to this rise in DFD methodology were Tom DeMarco, Chris Gane and Trish Sarson. They teamed up in different combinations to be the main definers of the symbols and notations used for a data flow diagram.

Symbol and Notations Used in DFDs

Two common systems of symbols are named after their creators:

- Yourdon and Coad
- Yourdon and DeMarco
- Gane and Sarson
One main difference in their symbols is that Yourdon-Coad and Yourdon-DeMarco use circles for processes, while Gane and Sarson use rectangles with rounded corners, sometimes called lozenges. There are other symbol variations in use as well, so the important thing to keep in mind is to be clear and consistent in the shapes and notations you use to communicate and collaborate with others.

Using any convention’s DFD rules or guidelines, the symbols depict the four components of data flow diagrams.

1. **External entity**: an outside system that sends or receives data, communicating with the system being diagrammed. They are the sources and destinations of information entering or leaving the system. They might be an outside organization or person, a computer system or a business system. They are also known as terminators, sources and sinks or actors. They are typically drawn on the edges of the diagram.
2. **Process**: any process that changes the data, producing an output. It might perform computations, or sort data based on logic, or direct the data flow based on business rules. A short label is used to describe the process, such as “Submit payment.”
3. **Data store**: files or repositories that hold information for later use, such as a database table or a membership form. Each data store receives a simple label, such as “Orders.”
4. **Data flow**: the route that data takes between the external entities, processes and data stores. It portrays the interface between the other components and is shown with arrows, typically labeled with a short data name, like “Billing details.”[13]

<table>
<thead>
<tr>
<th>Notation</th>
<th>Yourdon and Coad</th>
<th>Gane and Sarson</th>
</tr>
</thead>
<tbody>
<tr>
<td>External Entity</td>
<td>![Image]</td>
<td>![Image]</td>
</tr>
<tr>
<td>Process</td>
<td>![Image]</td>
<td>![Image]</td>
</tr>
<tr>
<td>Data Store</td>
<td>![Image]</td>
<td>![Image]</td>
</tr>
<tr>
<td>Data Flow</td>
<td>![Image]</td>
<td>![Image]</td>
</tr>
</tbody>
</table>

Figure 2.7 DFD Symbol.
2.9 ERD (Entity Relationship Diagram)

An Entity Relationship (ER) Diagram is a type of flowchart that illustrates how “entities” such as people, objects or concepts relate to each other within a system. ER Diagrams are most often used to design or debug relational databases in the fields of software engineering, business information systems, education and research. Also known as ERDs or ER Models, they use a defined set of symbols such as rectangles, diamonds, ovals and connecting lines to depict the interconnectedness of entities, relationships and their attributes. They mirror grammatical structure, with entities as nouns and relationships as verbs.

ER diagrams are related to data structure diagrams (DSDs), which focus on the relationships of elements within entities instead of relationships between entities themselves. ER diagrams also are often used in conjunction with data flow diagrams (DFDs), which map out the flow of information for processes or systems.

Uses of entity relationship diagrams

- **Database design**: ER diagrams are used to model and design relational databases, in terms of logic and business rules (in a logical data model) and in terms of the specific technology to be implemented (in a physical data model.) In software engineering, an ER diagram is often an initial step in determining requirements for an information systems project. It’s also later used to model a particular database or databases. A relational database has an equivalent relational table and can potentially be expressed that way as needed.

- **Database troubleshooting**: ER diagrams are used to analyze existing databases to find and resolve problems in logic or deployment. Drawing the diagram should reveal where it’s going wrong.

- **Business information systems**: The diagrams are used to design or analyze relational databases used in business processes. Any business process that uses fielded data involving entities, actions and interplay can potentially benefit from a relational database. It can streamline processes, uncover information more easily and improve results.

- **Business process re-engineering (BPR)**: ER diagrams help in analyzing databases used in business process re-engineering and in modeling a new database setup.
- **Education:** Databases are today’s method of storing relational information for educational purposes and later retrieval, so ER Diagrams can be valuable in planning those data structures.

- **Research:** Since so much research focuses on structured data, ER diagrams can play a key role in setting up useful databases to analyze the data.

### The Components and Features of an ER Diagram

ER Diagrams are composed of entities, relationships and attributes. They also depict cardinality, which defines relationships in terms of numbers. Here’s a glossary:

**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, car or product. Typically shown as a rectangle.

**Entity type:** A group of definable things, such as students or athletes, whereas the entity would be the specific student or athlete. Other examples: customers, cars or products.

**Entity set:** Same as an entity type, but defined at a particular point in time, such as students enrolled in a class on the first day. Other examples: Customers who purchased last month, cars currently registered in Florida. A related term is instance, in which the specific person or car would be an instance of the entity set.

**Entity categories:** Entities are categorized as strong, weak or associative. A strong entity can be defined solely by its own attributes, while a weak entity cannot. An associative entity associates entities (or elements) within an entity set.
**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.

![Relationship Symbol](image)

**Attribute:** A property or characteristic of an entity. Often shown as an oval or circle.

![Attribute Symbol](image)

Descriptive attribute: A property or characteristic of a relationship (versus of an entity.)

Attribute categories: Attributes are categorized as simple, composite, derived, as well as single-value or multi-value. Simple: Means the attribute value is atomic and can’t be further divided, such as a phone number. Composite: Sub-attributes spring from an attribute. Derived: Attributed is calculated or otherwise derived from another attribute, such as age from a birthdate.

![Derived Attribute Symbol](image)
Multi-value: More than one attribute value is denoted, such as multiple phone numbers for a person.

![Multivalued Attribute](image)

Figure 2.13 Multivalued Symbol.

Single-value: Just one attribute value. The types can be combined, such as: simple single-value attributes or composite multi-value attributes.

**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.

![Cardinality Symbol](image)

Figure 2.14 Cardinality Symbol.
Cardinality views: Cardinality can be shown as look-across or same-side, depending on where the symbols are shown.

Cardinality constraints: The minimum or maximum numbers that apply to a relationship.[14]

Relationship Between Entities

A relationship is how the data is shared between entities. There are three types of relationships between entities:

1. One-to-One

One instance of an entity (A) is associated with one other instance of another entity (B). For example, in a database of employees, each employee name (A) is associated with only one social security number (B).

![Figure 2.15 One to One Symbol.](image)

2. One-to-Many

One instance of an entity (A) is associated with zero, one or many instances of another entity (B), but for one instance of entity B there is only one instance of entity A. For example, for a company with all employees working in one building, the building name (A) is associated with many different employees (B), but those employees all share the same singular association with entity A.

![Figure 2.16 One to Many Symbol.](image)
3. Many-to-Many

One instance of an entity (A) is associated with one, zero or many instances of another entity (B), and one instance of entity B is associated with one, zero or many instances of entity A. For example, for a company in which all of its employees work on multiple projects, each instance of an employee (A) is associated with many instances of a project (B), and at the same time, each instance of a project (B) has multiple employees (A) associated with it.

![Figure 2.17 Many to Many Symbol.](image)

**ER Diagram Symbols**

In an ER diagram, symbols are commonly used to represent the types of information. Most diagrams will use the following:

- Boxes represent entities.
- Diamonds represent relationships
- Circles (ovals) represent attributes

2.10 SDLC

SDLC or the Software Development Life Cycle is a process that produces software with the highest quality and lowest cost in the shortest time. SDLC includes a detailed plan for how to develop, alter, maintain, and replace a software system.

SDLC involves several distinct stages, including planning, design, building, testing, and deployment. Popular SDLC models include the waterfall model, spiral model, and Agile model.

**Agile SDLC Model**

In the agile methodology after every development iteration, the customer is able to see the result and understand if he is satisfied with it or he is not. This is one of the advantages of the agile software development life cycle model. One of its disadvantages is that with the absence of defined requirements it is difficult to estimate the resources and development cost. Extreme programming
is one of the practical use of the agile model. The basis of such model consists of short weekly meetings – Sprints which are the part of the Scrum approach.[17]

Use cases for the agile model:

- The users’ needs change dynamically
- Less price for the changes implemented because of many iterations
- Unlike the waterfall mode, it requires only initial planning to start the project.

Advantages of Agile model:

- Customer satisfaction by rapid, continuous delivery of useful software.
- People and interactions are emphasized rather than process and tools. Customers, developers and testers constantly interact with each other.
- Working software is delivered frequently (weeks rather than months).
- Face-to-face conversation is the best form of communication.
- Close, daily cooperation between business people and developers.
- Continuous attention to technical excellence and good design.
- Regular adaptation to changing circumstances.
- Even late changes in requirements are welcomed

Disadvantages of Agile model:

- In case of some software deliverables, especially the large ones, it is difficult to assess the effort required at the beginning of the software development life cycle.
- There is lack of emphasis on necessary designing and documentation.
- The project can easily get taken off track if the customer representative is not clear what final outcome that they want.
- Only senior programmers are capable of taking the kind of decisions required during the development process. Hence it has no place for newbie programmers, unless combined with experienced resources.
When to use agile model:

- When new changes are needed to be implemented. The freedom agile gives to change is very important. New changes can be implemented at very little cost because of the frequency of new increments that are produced.

- To implement a new feature the developers need to lose only the work of a few days, or even only hours, to roll back and implement it. Unlike the waterfall model in agile model very limited planning is required to get started with the project. Agile assumes that the end users’ needs are ever changing in a dynamic business and IT world. Changes can be discussed and features can be newly effected or removed based on feedback. This effectively gives the customer the finished system they want or need.

- Both system developers and stakeholders alike, find they also get more freedom of time and options than if the software was developed in a more rigid sequential way. Having options gives them the ability to leave important decisions until more or better data or even entire hosting programs are available; meaning the project can continue to move forward without fear of reaching a sudden standstill.[16]