**Data Flow Diagram**

Data flow diagram is a graphical representation of data flow in an information system. It is capable of depicting incoming data flow, outgoing data flow and stored data. The DFD does not mention anything about how data flows through the system.

There is a prominent difference between DFD and Flowchart. The flowchart depicts flow of control in program modules. DFDs depict flow of data in the system at various levels. DFD does not contain any control or branch elements.

**Types of DFD**

Data Flow Diagrams are either Logical or Physical.

 **Logical DFD** - This type of DFD concentrates on the system process and flow of data in the system. For example in a Banking software system, how data is moved between different entities.

 **Physical DFD** - This type of DFD shows how the data flow is actually implemented in the system. It is more specific and close to the implementation.

*DEPT OF CSE & IT VSSUT, Burla*



**DFD Components**

DFD can represent Source, destination, storage and flow of data using the following set of components -

Fig 10.1: DFD Components

 **Entities** - Entities are source and destination of information data. Entities are represented by rectangles with their respective names.

 **Process** - Activities and action taken on the data are represented by Circle or Round-edged rectangles.

 **Data Storage** - There are two variants of data storage - it can either be represented as a rectangle with absence of both smaller sides or as an open-sided rectangle with only one side missing.

 **Data Flow** - Movement of data is shown by pointed arrows. Data movement is shown from the base of arrow as its source towards head of the arrow as destination.

**Importance of DFDs in a good software design**

The main reason why the DFD technique is so popular is probably because of the fact that DFD is a very simple formalism – it is simple to understand and use. Starting with a set of high-level functions that a system performs, a DFD model hierarchically represents various sub-functions. In fact, any hierarchical model is simple to understand. Human mind is such that it can easily understand any hierarchical model of a system – because in a hierarchical model, starting with a very simple and abstract model of a system, different details of the system are slowly introduced through different hierarchies. The data flow diagramming technique also follows a very simple set of intuitive concepts and rules. DFD is an elegant modeling technique that turns out to be useful not only to represent the results of structured analysis of a software problem, but also for several other applications such as showing the flow of documents or items in an organization. *DEPT OF CSE & IT VSSUT, Burla*

**Data Dictionary**

A data dictionary lists all data items appearing in the DFD model of a system. The data items listed include all data flows and the contents of all data stores appearing on the DFDs in the DFD model of a system. A data dictionary lists the purpose of all data items and the definition of all composite data items in terms of their component data items. For example, a data dictionary entry may represent that the data **grossPay** consists of the components regularPay and overtimePay.

**grossPay = regularPay + overtimePay**

For the smallest units of data items, the data dictionary lists their name and their type. Composite data items can be defined in terms of primitive data items using the following data definition operators:

**+**: denotes composition of two data items, e.g. **a+b** represents data a and **b**.

**[,,]**: represents selection, i.e. any one of the data items listed in the brackets can occur. For example, **[a,b]** represents either **a** occurs or **b** occurs.

**()**: the contents inside the bracket represent optional data which may or may not appear. e.g. **a+(b)** represents either **a** occurs or **a+b** occurs.

**{}**: represents iterative data definition, e.g. **{name}5** represents five **name** data. **{name}\*** represents zero or more instances of **name** data.

**=**: represents equivalence, e.g. **a=b+c** means that **a** represents **b** and **c**.

**/\* \*/**: Anything appearing within **/\*** and **\*/** is considered as a comment.

**Example 1:** Tic-Tac-Toe Computer Game

Tic-tac-toe is a computer game in which a human player and the computer make alternative moves on a 3×3 square. A move consists of marking previously unmarked square. The player who first places three consecutive marks along a straight line on the square (i.e. along a row, column, or diagonal) wins the game. As soon as either the human player or the computer wins, a message congratulating the winner should be displayed. If neither player manages to get three consecutive marks along a straight line, but all the squares on the board are filled up, then the game is drawn. The computer always tries to win a game. *DEPT OF CSE & IT VSSUT, Burla*





**(a)**

Fig 10.2 (a) Level 0 (b) Level 1 DFD for Tic-Tac-Toe game *DEPT OF CSE & IT VSSUT, Burla*

It may be recalled that the DFD model of a system typically consists of several DFDs: level 0, level 1, etc. However, a single data dictionary should capture all the data appearing in all the DFDs constituting the model. Figure 10.2 represents the level 0 and level 1 DFDs for the tic-tac-toe game. The data dictionary for the model is given below.

**Data Dictionary for the DFD model in Example 1**

move: integer /\*number between 1 and 9 \*/

display: game+result

game: board

board: {integer}9

result: [“computer won”, “human won” “draw”]

**Importance of Data Dictionary**

A data dictionary plays a very important role in any software development process because of the following reasons:

• A data dictionary provides a standard terminology for all relevant data for use by the engineers working in a project. A consistent vocabulary for data items is very important, since in large projects different engineers of the project have a tendency to use different terms to refer to the same data, which unnecessary causes confusion.

• The data dictionary provides the analyst with a means to determine the definition of different data structures in terms of their component elements.

**UML Diagrams**

UML can be used to construct nine different types of diagrams to capture five different views of a system. Just as a building can be modeled from several views (or perspectives) such as ventilation perspective, electrical perspective, lighting perspective, heating perspective, etc.; the different UML diagrams provide different perspectives of the software system to be developed and facilitate a comprehensive understanding of the system. Such models can be refined to get the actual implementation of the system.

The UML diagrams can capture the following five views of a system**:**

• User’s view

• Structural view

• Behavioral view

• Implementation view

• Environmental view

Fig. 12.1: Different types of diagrams and views supported in UML

**User’s view:** This view defines the functionalities (facilities) made available by the system to its users. The users’ view captures the external users’ view of the system in terms of the functionalities offered by the system. The users’ view is a black-box view of the system where the internal structure, the dynamic behavior of different system components, the implementation etc. are not visible. The users’ view is very different from all other views in the sense that it is a functional model compared to the object model of all other views. The users’ view can be considered as the central view and all other views are expected to conform to this view. This thinking is in fact the crux of any user centric development style.

**Structural view:** The structural view defines the kinds of objects (classes) important to the understanding of the working of a system and to its implementation. It also captures the

*DEPT OF CSE & IT VSSUT, Burla*

relationships among the classes (objects). The structural model is also called the static model, since the structure of a system does not change with time.

**Behavioral view:** The behavioral view captures how objects interact with each other to realize the system behavior. The system behavior captures the time-dependent (dynamic) behavior of the system.

**Implementation view:** This view captures the important components of the system and their dependencies.

**Environmental view:** This view models how the different components are implemented on different pieces of hardware.

------------------------------Metrics---------------------------------

*Software metrics are quantifiable measures that could be used to measure different*

*characteristics of a software system or the software-development process.*

Metrics and measurements are necessary aspects of managing a softwaredevelopment

project. For effective monitoring, management needs to get information

about the project: how far it has progressed, how much development has taken

place, how far behind schedule it is, and the quality of the development so far.

Based on this information, decisions can be made about the project. Without

proper metrics to quantify the required information, subjective opinion would

have to be used, which is often unreliable and goes against the fundamental goals

of engineering. Hence, we can say that metrics-based management is also a key

component in the software-engineering strategy to achieve its objectives.

**6.1.1 Definition**

Software metrics can be defined as “*The continuous application of measurement-based*

*techniques to the software-development process and its products to supply meaningful and*

**149**

**150** SOFTWARE ENGINEERING AND TESTING

*timely management information, together with the use of those techniques to improve that*

*process and its products*.”

Metrics ensure that the final product is of high quality and the productivity

of the project stays high. In this sequence metrics for intermediate products of

requirements and design is to predict or get some idea about the metrics of the final

product. Several metrics have been designed for coding; namely, size, complexity,

style, and reliability.

**6.1.2 Categories of Metrics**

There are three categories of software metrics, which are as follows:

1. ***Product Metrics*.** Product metrics describe the characteristics of the product,

such as size, complexity, design features, performance, efficiency, reliability,

portability, etc.

2. ***Process Metrics*.** Process metrics describe the effectiveness and quality of the

processes that produce the software product. Examples are:

\_ Effort required in the process

\_ Time to produce the product

\_ Effectiveness of defect removal during development

\_ Number of defects found during testing

\_ Maturity of the process

3. ***Project Metrics*.** Project metrics describe the project characteristics and

execution. Examples are:

\_ Number of software developers

\_ Staffing pattern over the life-cycle of the software

\_ Cost and schedule

\_ Productivity

**6.1.3 Attributes of Effective Software Metrics**

Hundreds of metrics have been proposed for computer software, but not all

provide practical support to the software engineer. Some demand measurement

that is too complex, others are so esoteric that few real-world professionals have

any hope of understanding them, and others violate the basic intuitive notions of

what high-quality software really is.

Ejiogu defines a set of attributes that should be encompassed by effective

software metrics. The derived metric and the measures that lead to it should be:

SOFTWARE MEASUREMENT AND METRICS **151**

\_ *Simple and computable:* It should be relatively easy to learn how to derive the

metric, and its computation should not demand inordinate effort or time.

\_ *Empirically and intuitively persuasive:* The metrics should satisfy the engineer’s

intuitive notions about the product attribute under consideration.

\_ *Consistent and objective:* The metric should always yield results that are

unambiguous.

\_ *Consistent in the use of units and dimensions:* The mathematical computation of

the metric should use measures that do not lead to bizarre combinations of

units.

\_ *Programming-language independent:* Metrics should be based on the analysis

model, the design model, or the structure of the program itself.

\_ *An effective mechanism for high-quality feedback:* That is, the metric should lead to

a higher-quality end product.

Software Requirement Specification (SRS)

A **software requirements specification**(SRS) is a detailed description of a software system to be developed with its functional and non-functional requirements. The SRS is developed based the agreement between customer and contractors. It may include the use cases of how user is going to interact with software system. The software requirement specification document consistent of all necessary requirements required for project development. To develop the software system we should have clear understanding of Software system. To achieve this we need to continuous communication with customers to gather all requirements.

A good SRS defines the how Software System will interact with all internal modules, hardware, communication with other programs and human user interactions with wide range of real life scenarios. Using the *Software requirements specification* (SRS) document on QA lead, managers creates test plan. It is very important that testers must be cleared with every detail specified in this document in order to avoid faults in test cases and its expected results.

It is highly recommended to review or test SRS documents before start writing test cases and making any plan for testing. Let’s see how to test SRS and the important point to keep in mind while testing it.

**1. Correctness of SRS should be checked.**Since the whole testing phase is dependent on SRS, it is very important to check its correctness. There are some standards with which we can compare and verify.

**2. Ambiguity should be avoided.**Sometimes in SRS, some words have more than one meaning and this might confused testers making it difficult to get the exact reference. It is advisable to check for such ambiguous words and make the meaning clear for better understanding.

**3. Requirements should be complete.**When tester writes test cases, what exactly is required from the application, is the first thing which needs to be clear. For e.g. if application needs to send the specific data of some specific size then it should be clearly mentioned in SRS that how much data and what is the size limit to send.

**4. Consistent requirements.**The SRS should be consistent within itself and consistent to its reference documents. If you call an input “Start and Stop” in one place, don’t call it “Start/Stop” in another. This sets the standard and should be followed throughout the testing phase.

**5. Verification of expected result:**SRS should not have statements like “Work as expected”, it should be clearly stated that what is expected since different testers would have different thinking aspects and may draw different results from this statement.

**6. Testing environment:** some applications need specific conditions to test and also a particular environment for accurate result. SRS should have clear documentation on what type of environment is needed to set up.

**7. Pre-conditions defined clearly:**one of the most important part of test cases is pre-conditions. If they are not met properly then actual result will always be different expected result. Verify that in SRS, all the pre-conditions are mentioned clearly.

**8. Requirements ID:** these are the base of test case template. Based on requirement Ids, test case ids are written. Also, requirements ids make it easy to categorize modules so just by looking at them, tester will know which module to refer. SRS must have them such as id defines a particular module.

**9. Security and Performance criteria:**security is priority when a software is tested especially when it is built in such a way that it contains some crucial information when leaked can cause harm to business. Tester should check that all the security related requirements are properly defined and are clear to him. Also, when we talk about performance of a software, it plays a very important role in business so all the requirements related to performance must be clear to the tester and he must also know when and how much stress or load testing should be done to test the performance.

**10. Assumption should be avoided:**sometimes when requirement is not cleared to tester, he tends to make some assumptions related to it, which is not a right way to do testing as assumptions could go wrong and hence, test results may vary. It is better to avoid assumptions and ask clients about all the “missing requirements” to have a better understanding of expected results.

**11. Deletion of irrelevant requirements:**there are more than one team who work on SRS so it might be possible that some irrelevant requirements are included in SRS. Based on the understanding of the software, tester can find out which are these requirements and remove them to avoid confusions and reduce work load.

**12. Freeze requirements:** when an ambiguous or incomplete requirement is sent to client to analyze and tester gets a reply, that requirement result will be updated in the next SRS version and client will freeze that requirement. Freezing here means that result will not change again until and unless some major addition or modification is introduced in the software.

Most of the defects which we find during testing are because of either incomplete requirements or ambiguity in SRS. To avoid such defects it is very important to test software requirements specification before writing the test cases. Keep the latest version of SRS with you for reference and keep yourself updated with the latest change made to the SRS. Best practice is to go through the document very carefully and note down all the confusions, assumptions and incomplete requirements and then have a meeting with the client to get them clear before development phase starts as it becomes costly to fix the bugs after the software is developed. After all the requirements are cleared to a tester, it becomes easy for him to write effective test cases and accurate expected results.

Over to you: I think I have addressed all major points of Software requirements specification. Have you ever worked on “**Testing of software requirements specification (SRS) document**”. If yes then please make sure that you share it with your QA friends. Please leave your questions/tips/suggestions in the comment section below and I’ll try to answer as many as I can.