**Requirement Analysis and Specifications**

Table of Contents

[SRS Document 3](#_Toc83393036)

[Requirement Gathering Process 4](#_Toc83393037)

[Problem Analysis 6](#_Toc83393038)

[Requirement Analysis 7](#_Toc83393039)

[Business Process Modelling Notation 7](#_Toc83393040)

[Unified Modelling Language 8](#_Toc83393041)

[Flowcharts 9](#_Toc83393042)

[Data Flow Diagrams 9](#_Toc83393043)

[Other Approaches 10](#_Toc83393044)

[Specifications 11](#_Toc83393045)

[SRS Components 12](#_Toc83393046)

[SRS Characteristics 13](#_Toc83393047)

[Standard SRS Contents 14](#_Toc83393048)

[Requirements Validation 15](#_Toc83393049)

If we want to develop any software, the first thing we need to do is figure out what the **requirements** are. Without a clear set of requirements, it will be nearly impossible to actually create the software the customer is expecting.

A requirement is a condition or capability that must be possessed by the system. To get a requirement, we need to take what the user has in mind and extract a precise statement about what the future system will do. The process is rather difficult for several reasons:

* A future system is difficult to visualize, especially if similar applications do not exist.
* Clients are sometimes unable to specify what exactly they want from the system without seeing at least a prototype first.
* Client requirements change over time.

It is however, essential that we do a proper analysis of the requirements. This is done using several techniques which we will explore soon.

The output of the requirements analysis phase is the **Software Requirements Specification** (SRS) report. It contains the complete specifications of what the proposed system should do. It is intended for several groups of people:

* The developers will use it to understand what they need to create
* The testing team will use it to ensure that the developed software matches what the client’s requirements were
* The client’s organization will use it to check that the requirements they initially gave are present in the software
* The project manager will use it to create things like the product backlog

## SRS Document

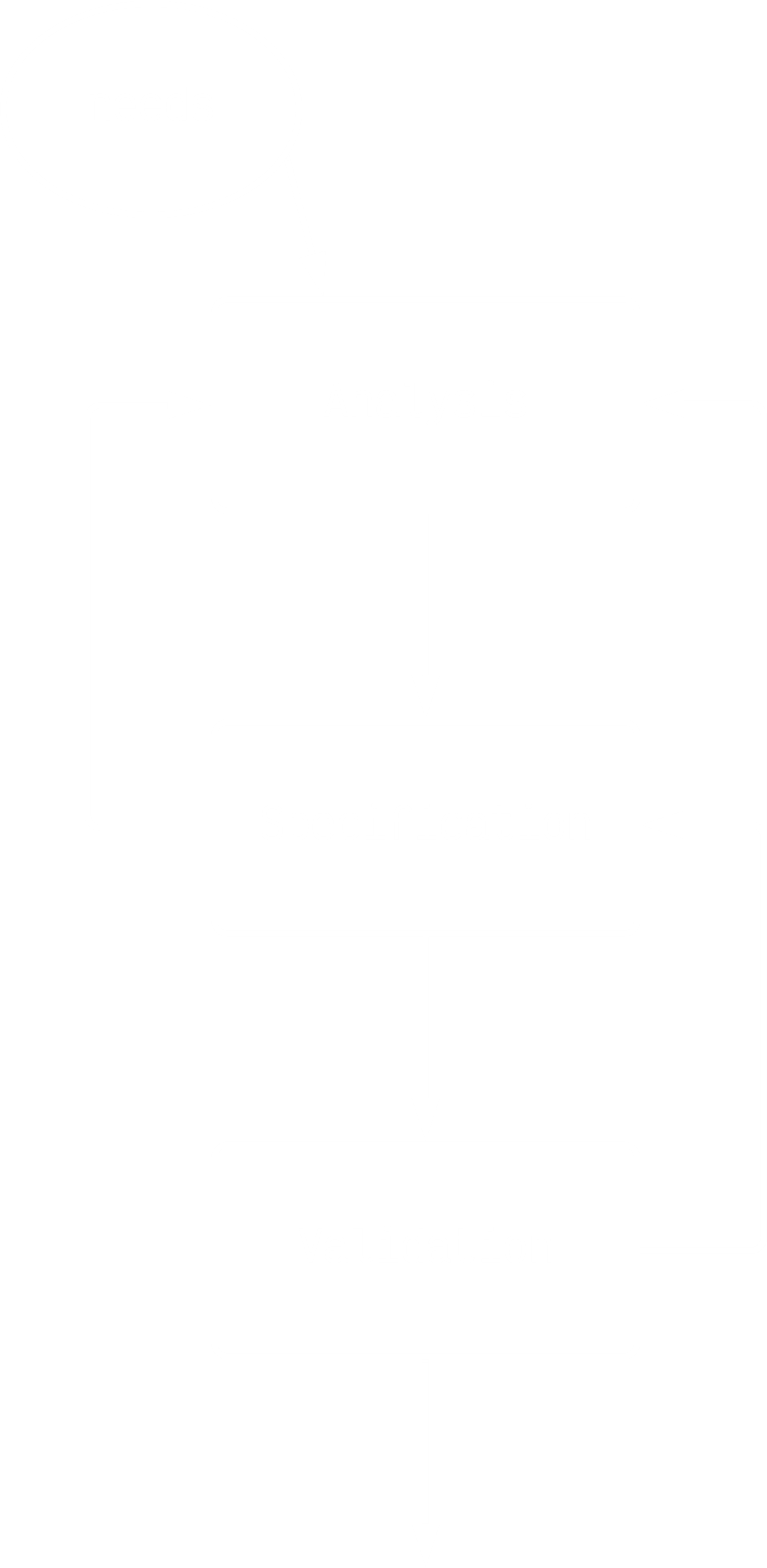
The SRS Document establishes the **basis of agreement** between the user and the developers. The user’s needs need to be satisfied, but they may not understand software and may thus be unable to explain in technical terms what they need. Sometimes, they may not even know what they want. The developers can develop software, but without technical terminology and specific requirements, they may not understand what they need to develop. The SRS document bridges this communication gap. It specifies the user’s needs in a way that both parties are able to understand.

The SRS is also used for **validation**. Once the software is built, we sit with the SRS and match the program with the SRS function by function. The functions mentioned in the SRS also have the inputs and outputs, so we can check everything.

To produce high quality software, it is essential to have a high-quality SRS. Without a proper SRS, **requirement errors** are seen in the final software. This is very **costly** to fix, since not even testers will be able to tell if the SRS itself is wrong. To avoid this, we need to communicate with the client and make sure the SRS shows exactly what they want, to the extent that they are even required to sign it. A good SRS can **minimize changes** and errors.

Clearly the SRS is important. However, in an agile development methodology, like Scrum, we do not really have an SRS. Instead, some form of business document is maintained that has all the user’s requirements. In the case of Scrum, these requirements are in the form of user stories. Each user story will later be converted to one or more product backlog items by the product owner.

## Requirement Gathering Process



The diagram above shows the steps we go through when gathering requirements. We take the **user needs** and **analyse** them. From our analysis, we draw out the **specifications**. We then need to **validate** these specifications.

The steps are simple enough but in practice, it can get very messy. The process is not straightforward. The steps can require repeated execution in an iterative manner, they could be executed in parallel and they could even overlap one another. Once we analyse user needs once and find some specifications, those specifications themselves could help with further analysis. Validation can reveal gaps and lead to even more analysis and drawing of specifications.

To deal with all of this, we could take a **modular approach**. Essentially, we try to break down whatever the user is saying into small parts and find the relationships between the parts. This makes it easier to understand the parts and spot any misunderstandings.

When gathering the requirements, we will end up with a huge amount of information. We need to be able to **organize** this information in the SRS. The better we are able to organize it all, the better we have understood the requirements. Techniques like data flow diagrams, object diagrams, etc. help here.

## Problem Analysis

In the analysis step, we are trying to gain an understanding of the needs, requirements and constraints of the software.

Analysis may involve:

* **Interviews** with the clients and users
* Studying **manuals**
* Studying **existing systems**
* Helping clients and users recognize **new possibilities**

Essentially, it is like becoming a consultant. We need to understand how the organization, client or user has structured their work and how the software to be developed will fit into all of this.

There are some issues we are likely to face in the analysis stage:

* Obtaining the **necessary information**, since mostly, the user does not really know what they want.
* **Organizing** the gathered information and discarding any repeated information, since there can be a huge amount of information.
* Ensuring **completeness**, i.e. ensuring all the requirements are gathered.
* Ensuring **consistency**, i.e. ensuring the requirements do not contradict one another.

To deal with this, again, we **partition** the problem. The partitioning could be in reference to objects in the system, functions or events in the system.

## Requirement Analysis

From the information we gathered, we need to figure out what exactly the requirements are. There is no defined methodology to obtaining the information, but there are a few techniques available that are generally used to figure out the business workflow so we can analyse, understand and make changes to the workflow.

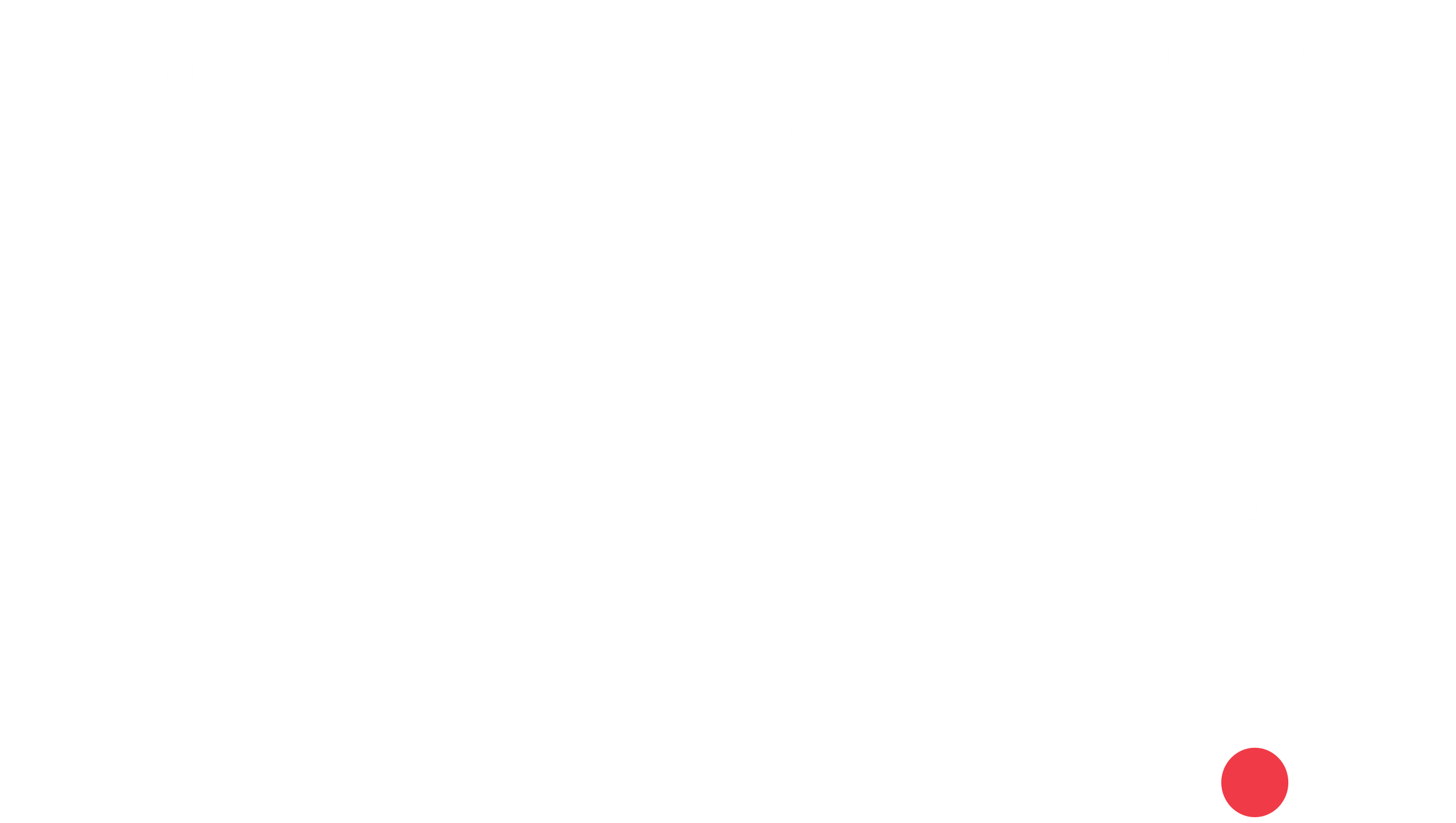
### Business Process Modelling Notation

A **BPMN** shows a graphical representation of a particular **business process** using simple objects. This helps the organization communicate in a standard manner.

A BPMN may include:

* Flow objects
* Connection objects
* Swim lanes
* Artifacts

Consider this diagram:



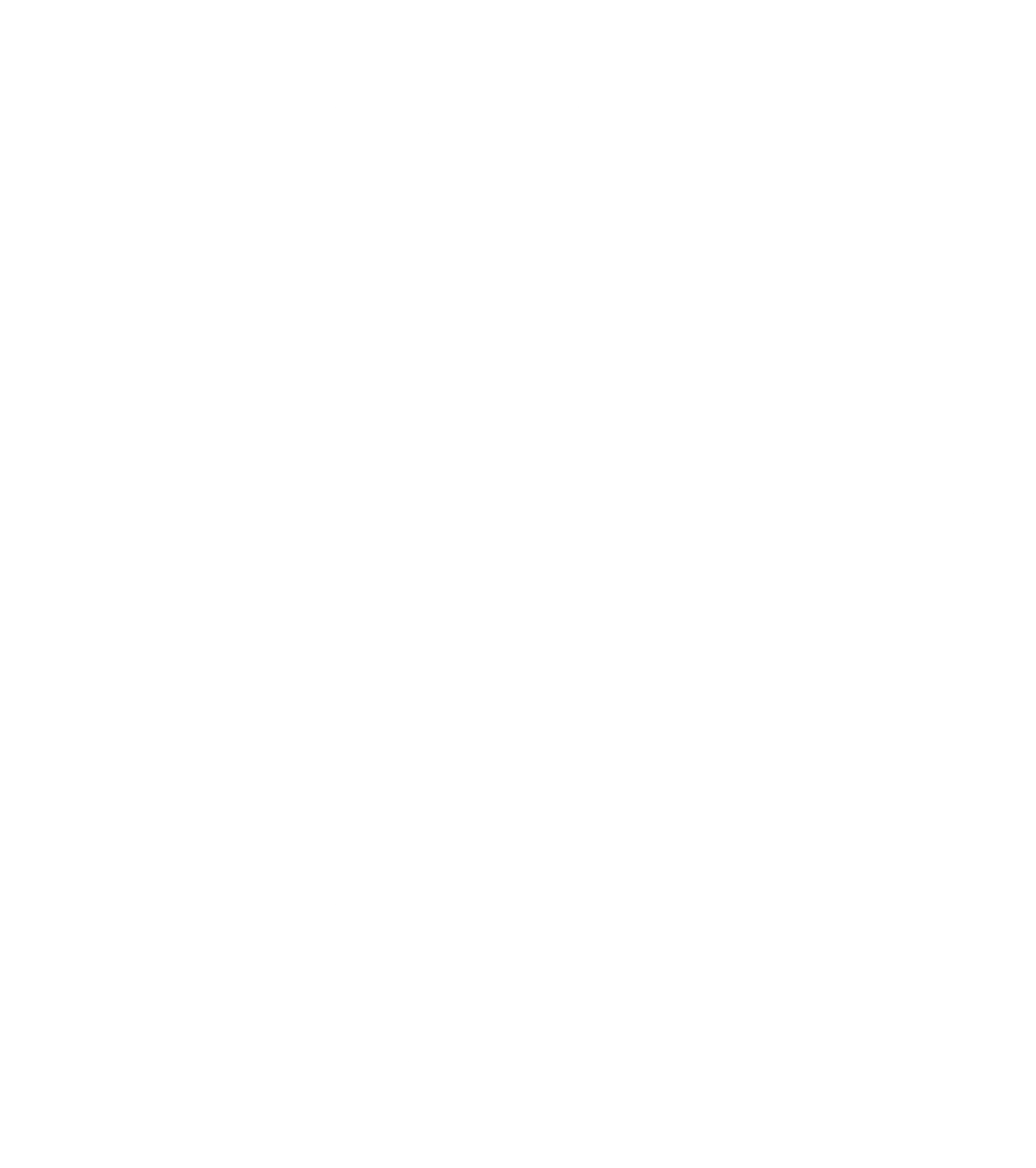
All we are doing here is, when the stated time and date has been reached, we check if there is a working group actively working, and if they are, we send them a list of issues. This is just one specific process and could be part of some software we need to build. There could be lots of BPMN diagrams like this.

A BPMN model needs to give details about **who** is performing each of the activities in the diagram and what **data elements** are required.

### Unified Modelling Language

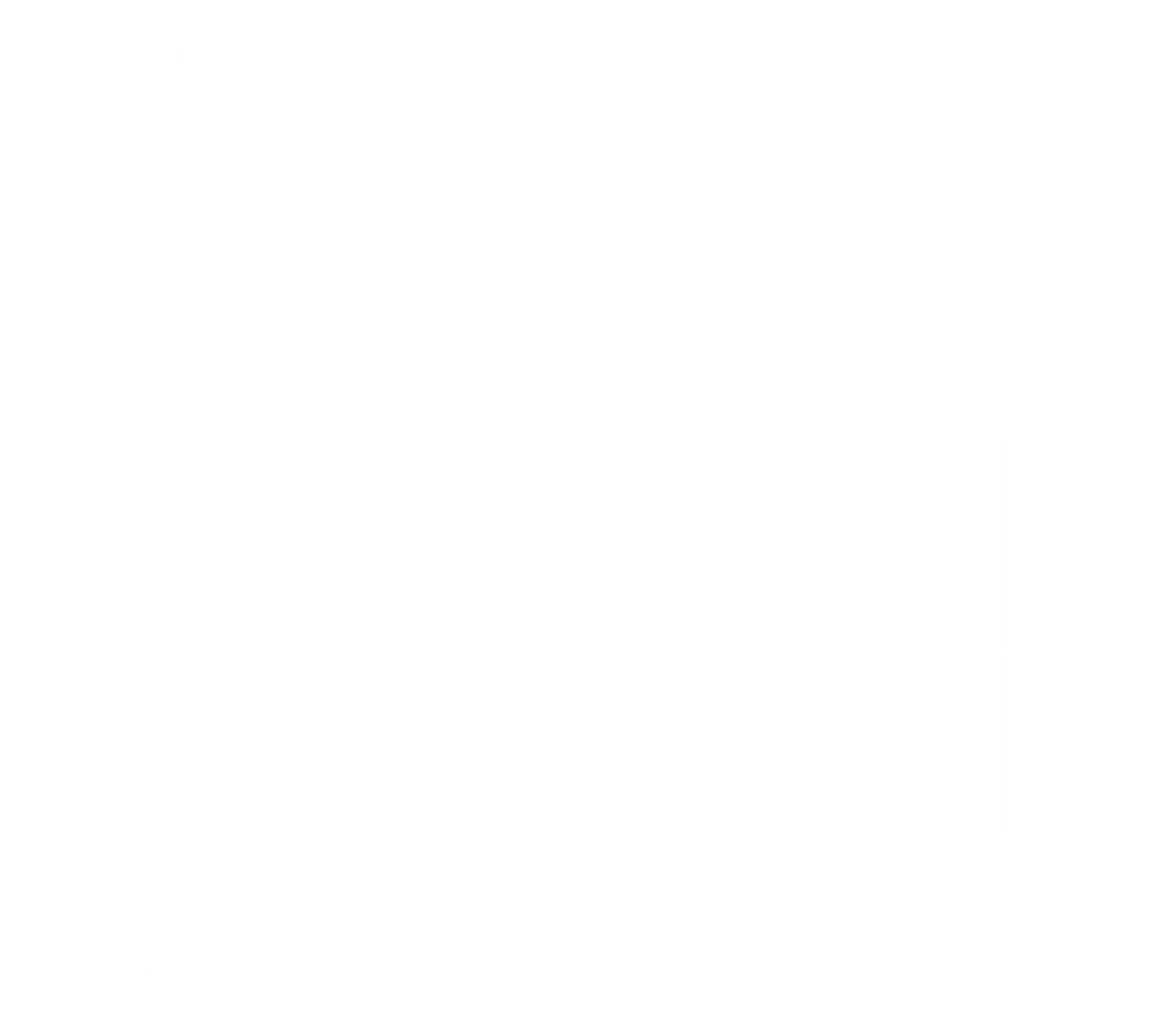
**UML** is a modelling standard used for specification, development, visualization and documentation of a software system. Essentially, it shows the steps the **actual software** goes through. It uses things like:

* States
* Objects
* Activities
* Class Diagrams



### Flowcharts

A **flowchart** shows the connections between different activities and how we go from one to another.

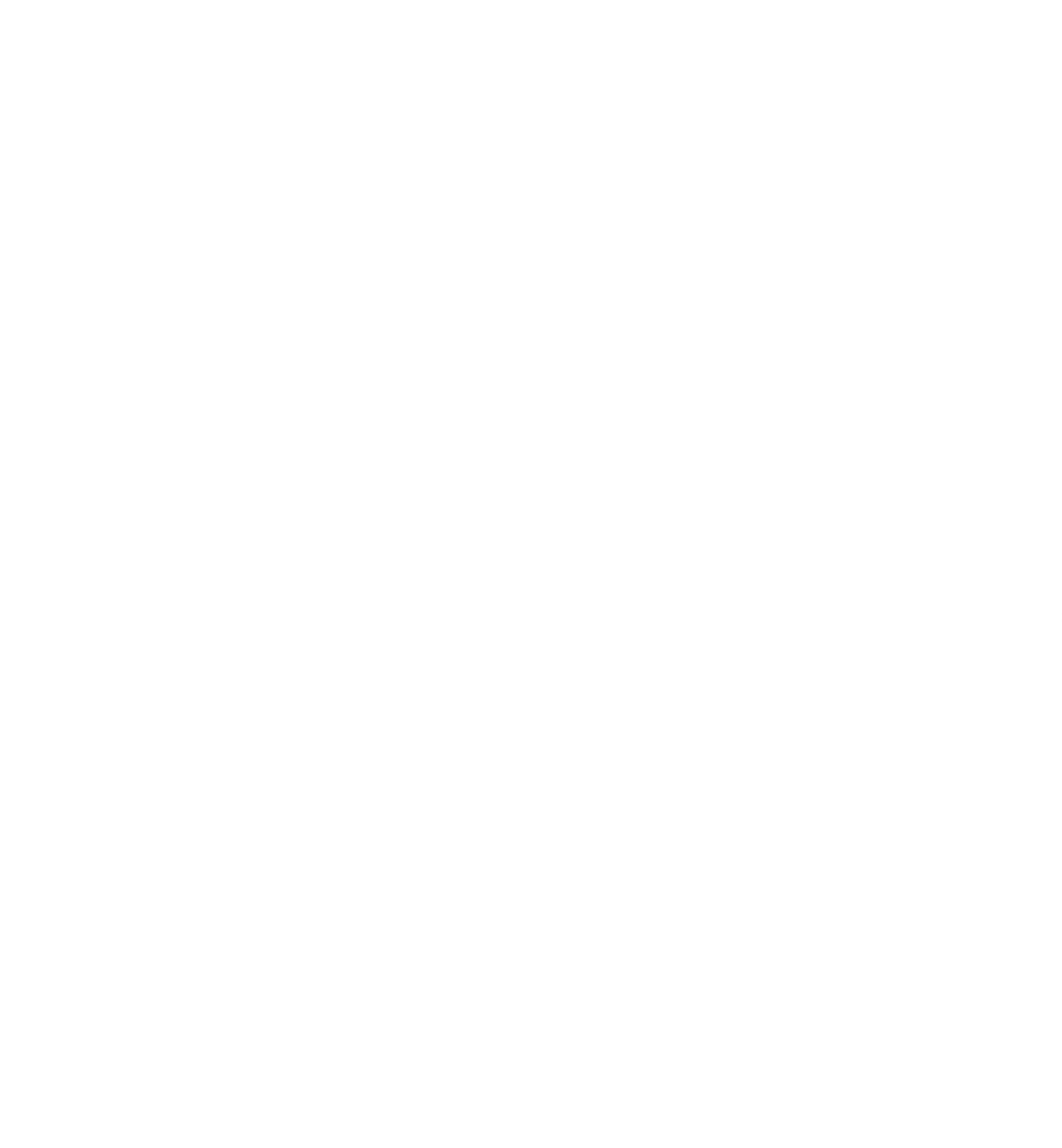


### Data Flow Diagrams

A **DFD** shows us how **data flows** through the system. The four parts of a DFD are:

* Processes
* Data Flows
* Data Stores
* External Entities

A DFD shows the system as a way to transform inputs into outputs.



There are some rules we need to follow when creating a DFD:

* Every process needs to have at least one input and output.
* Every data store needs to have at least one data flow in and one data flow out.
* Data stored in the system must go through a process.
* All processes in a DFD must have data flows to another process or data store.

### Other Approaches

Other approaches to requirement analysis include **prototyping** and **object-oriented design**. It is not normal to use these, but for very complex projects, it can be easier to just define classes, methods, attributes, class hierarchies and associations between classes.

## Specifications

The final output of requirement analysis is the **specifications**, i.e. the SRS document. None of the diagrams we saw above are part of the SRS. This is because those diagrams concentrated on how the system would work internally, but the SRS concentrates on the visible behaviour of the system.

For example, a DFD could show us how a new user signing up for the system is handled by the sign-up process. However, the fact that the sign-up feature must be available 24/7 is not shown there. This is something that will be specified in the SRS.

To reiterate, the requirements we gather are not the specifications. Requirements ensure that the customer and supplier are on the same page. Specifications are exactly what the software will do.

## SRS Components

If we can clarify what an SRS should contain, it will help us ensure that it is complete.

An SRS must specify the following types of requirements:

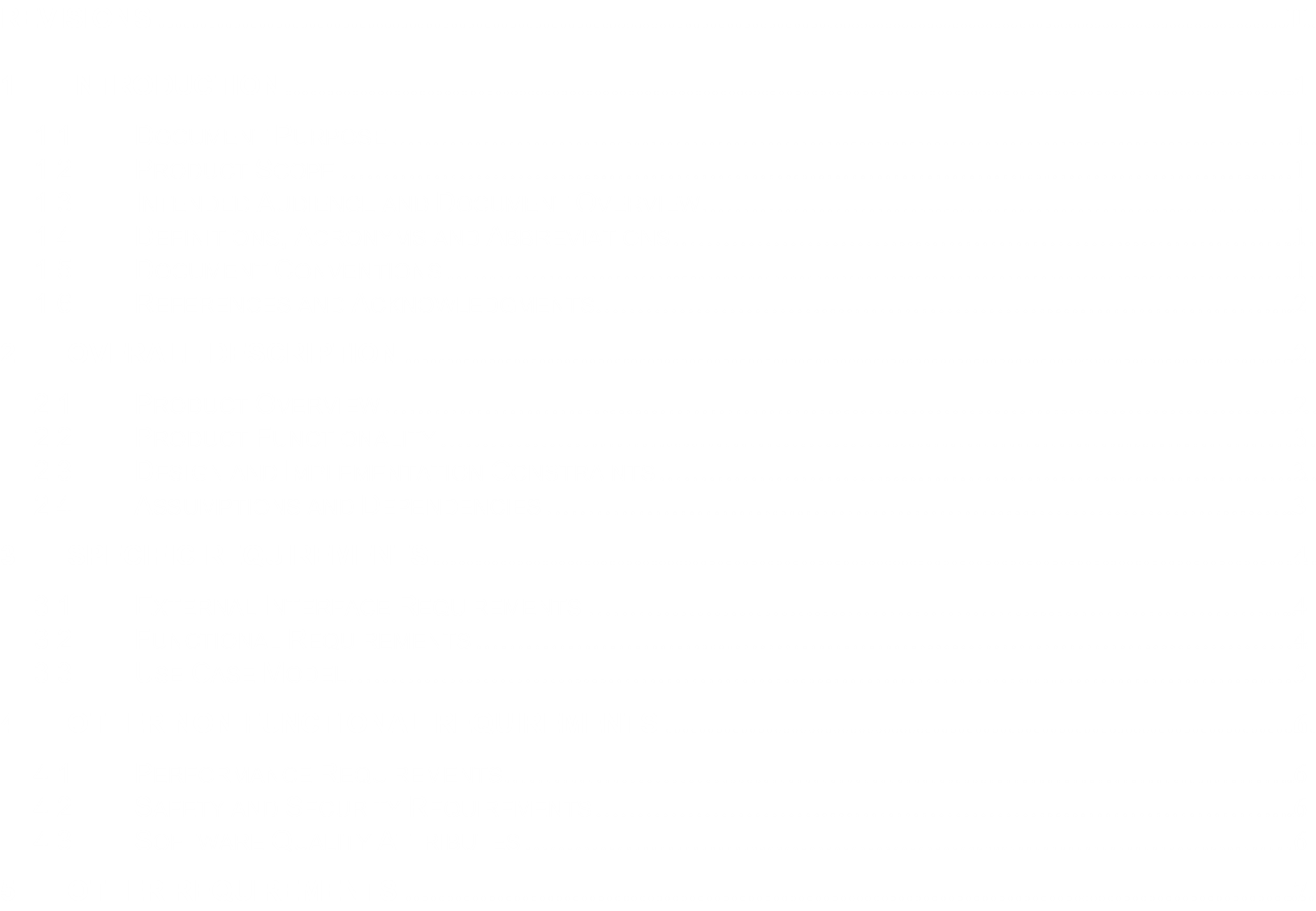
* **Functional** – An SRS should specify all the functionality that the system needs to support. We need to state what the different inputs are, what the corresponding outputs are and the relationships between them. We need to specify every operation that the system will perform and specify behaviour for invalid inputs.
* **Performance** – Performance requirements are also called non-functional requirements. This includes thinks like response time, throughput, capacity requirements, etc.
* **Design Constraints** – These are factors in the client’s environment that restrict the choices we have when developing the software. We need to include any standards stated by the client, requirements for compatibility with other systems, minimum reliability security requirements, backup requirements, fault tolerance requirements and hardware limitations.
* **External Interfaces** – This refers to any interactions that the program will have with people, hardware or software external to the program. Among these, the user interface is perhaps the most important. We need to specify exactly how these interactions will be handled by the system.

### SRS Characteristics

* **Correctness** – Each requirement in the SRS must accurately represent some feature in the final system.
* **Completeness** – All the desired features and characteristics must be specified. This is strongly related to correctness.
* **Unambiguousness** – Requirements must be stated in a way such that different people do not interpret their meanings differently. Otherwise, errors will crop up. This is extremely important, since the SRS is not a technical document, which makes it difficult to ensure it is unambiguous.
* **Verifiability** – There needs to be a cost-effective way to crosscheck the final software against the SRS.
* **Consistency** – Requirements in the SRS should not contradict each other or repeat each other.
* **Ranked by Importance or Stability** – This helps prioritize the features and reduces the risks presented by changing requirements.
* **Traceability** – The origins of each requirement and how they related to software elements should be clear. The business requirement that each functional requirement is meant to satisfy should be clear. It is possible that a single business requirement has multiple functional requirements. We also specify which tests are for which functional requirements. All of this information is included in a ‘traceability matrix’. Thus, if we find a bug in a particular test case, we will know exactly which functional requirement is faulty and which business requirement will not work properly due to the bug.

### Standard SRS Contents

A typical SRS document tends to look like this:



## Requirements Validation

The **validation** stage is important because there is always a lot of room for misunderstanding, which may cause errors that are expensive to fix. We need to try to remove as many of these errors that we can.

To validate the SRS, it is reviewed. This review is done by a team of people which includes:

* The author
* The client
* The user
* Representatives from the development team

The client and some users are essential here.

Standard inspection methods are used, where everyone goes through the document and discusses each point.

The validation stage is extremely effective. 40 – 80% of the errors can be caught in this stage.