Steps in building robust **software. (Characteristics for Good Design)**

**Design** This allows us to test for quality.

First, we need to understand the **requirements.**

**Implement** all requirements  
-Explicit (Need): Functional Requirements: Expected requirements for the system.  
-Implicit (Want): Non-Functional Requirements: Not expected, but adds value to the system (system still needs these). It can kind of be assumed that these will be there.  
  
Readable and understandable guide  
-A comprehensive guide is **needed** for those who test/use your software.   
-How to use the system. How the flow works.

Provide complete picture  
-Having a high level of extraction, which can be traced to specific systems.  
-**Data**: What is the information that foes in the system   
-**Functional:** What must the system do with the inputted data/interaction. How that data is interacted with.  
-**Behavioral Domains:** What you expect the system to produce from that functionality. How different parts of the system are supposed to react.

(The below was said before the above slide… might be in test… find in txtbook)

**Data:** How does the system communicate. How does data flow from one end to the other. (beginning to end)

**Interface**: **This** is extremely important, as software which does not look good, or is hard to use, will never be used.  
-A **good** way to test this is give it to **someone** who **hasn’t** **used** it before to test it.

**Components/Functions:** How does the … interact with each other.

**Quality Guidelines (Find these 8 guidelines in slide)**

This is indirectly related to **system design**.

Requires time, consistency. Not done over night.

What one perceives as quality, is **subjective**.   
-To combat this, we have **industry standards.  
-**Recognizable Styles and patterns: eg Waterfall  
Implemented in evolutionary manner: Change is good, everything is evolving/growing.

Lead to appropriate data structures: Document in such a way that we can trace back to the data

Interfaces that reduce complexity: To make things easier/simplistic

Derived from repeatable methods: from information

**Design Model (NB IN ST2 and Exam)**

We have **4** steps

Component:  
Interface:  
Architecture:  
Data:

**Data**:

Data vs Information: Data is unprocessed/raw facts. Information is processed data, with meaningful information extracted.  
Data: Receive from the system, about something on the system.  
Information: Translate the data, into useful information (for our project)

We must look at how we represent this information.  
Direct vs Indirect.  
It must be easily understandable by the user. Ie tables are terrible for this.

Look at the continuity of the data. What data am I inputting, and getting out of the system, and how can it be improved. What data can be represented/needed for the system to grow.

**Architecture (NB in st2 + exam)**

Major structural elements of the software and their relationships and defining those.

What patterns can be used to achieve the requirements in a specific domain.

The overall view of the solution. From core level.  
-how do the classes talk to eachother.

**Interface: (NB** Must be able to identify components in system, and explain them **Eg external interfaces)**

It is important to note that how nice a user interface looks, is subjective.

Most important element. How the software flows/looks, is extremely important

Internal interface: how the components interact with each other  
External interface: How this component talks with database. How does the user interact with system.  
User interface:

**Component**

How they (elements\_ behave with one another.

**Design Concepts (must know NB for exam/st2. Eg know procedural vs data, and an eg)**

These are the foundational building blocks to good relationships:

(**NB know 2 st**

**Abstraction**: Main goal is to hide complexity/unnecessary details.  
-Allows users to implement more complex details using class, without understanding how it works.  
**(sequence level: steps)-Procedural abstraction:** The natural steps (procedures) to do a function. Does not have to be specifically defined.  
**(Data: actual data in step)-Data Abstraction:** The actual collection of data that describes the object.

**Architecture**

Overall structure if the system and ways in which the structure provides conceptual integrity to the system

**Structural Properties**

**Extra functional properties:** Performance, capacity, reliability…

**Patterns**

A design structure that solves a particular design problem.  
-within a specific context.  
-amid other issues that might have an impact.

**Separation of Concerns**

Any complex problem can be handled if it can be broken down into smaller, simpler pieces that can be handled separately.

**Aspects**

**Modularity**

How many parts should the system be broken down into.

Too broken down? All you end out with is features, and no system.  
The more you break your system down, the more work you will need to do.

**Functional Independence**

Having a module that does not rely on other modules to work.  
-single minded functions.  
-Easy cohesion as no need for other modules to work for it to work.

**Refinement**

**Refactoring**

**Design Classes**

**Complete and sufficient: complete encapsulation of class.  
……**