# V-Model

The V-model is a type of SDLC model where process executes in a sequential manner in V-shape. It is also known as Verification and Validation model. It is based on the association of a testing phase for each corresponding development stage. Development of each step directly associated with the testing phase. The next phase starts only after completion of the previous phase i.e. for each development activity, there is a testing activity corresponding to it.



**Design Phase:**

* **Requirement Analysis:** This phase contains detailed communication with the customer to understand their requirements and expectations. This stage is known as Requirement Gathering.
* **System Design:** This phase contains the system design and the complete hardware and communication setup for developing product.
* **Architectural Design:** System design is broken down further into modules taking up different functionalities. The data transfer and communication between the internal modules and with the outside world (other systems) is clearly understood.
* **Module Design:** In this phase the system breaks down into small modules. The detailed design of modules is specified, also known as Low-Level Design (LLD).

**Testing Phases:**

* **Unit Testing:** Unit Test Plans are developed during module design phase. These Unit Test Plans are executed to eliminate bugs at code or unit level.
* **Integration testing:** After completion of unit testing Integration testing is performed. In integration testing, the modules are integrated and the system is tested. Integration testing is performed on the Architecture design phase. This test verifies the communication of modules among themselves.
* **System Testing:** System testing test the complete application with its functionality, inter dependency, and communication. It tests the functional and non-functional requirements of the developed application.
* **User Acceptance Testing (UAT):** UAT is performed in a user environment that resembles the production environment. UAT verifies that the delivered system meets user’s requirement and system is ready for use in real world.

# Static Code Analysis

Static analysis or Static Code Analysis, is a way of automatically analyzing code without executing it. SonarCloud is a cloud offering of SonarQube app. It is Free for Open source projects.

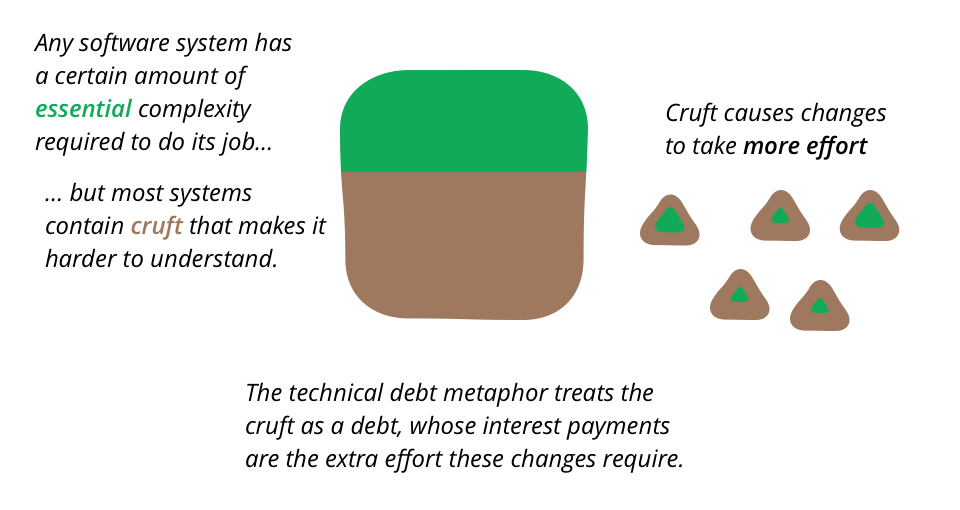
Static Code Analysis is a method of computer program debugging that is done by examining the code without executing the program. The process provides an understanding of the code structure, and can help to ensure that the code adheres to industry standards. Automated tools can assist programmers and developers in carrying out static analysis. The process of scrutinizing code by visual inspection alone (by looking at a printout, for example), without the assistance of automated tools, is sometimes called program understanding or program comprehension.

The principal advantage of static analysis is the fact that it can reveal errors that do not manifest themselves until a disaster occurs weeks, months or years after release. Nevertheless, static analysis is only a first step in a comprehensive software quality-control regime. After static analysis has been done, dynamic analysis is often performed in an effort to uncover subtle defects or vulnerabilities. In computer terminology, static means fixed, while dynamic means capable of action and/or change. Dynamic analysis involves the testing and evaluation of a program based on execution. Static and dynamic analysis, considered together, are sometimes referred to as glass-box testing.

# Technical Debt

**Technical debt** (*also known as****design debt****or****code debt***) is a concept in software development that reflects the implied cost of additional rework caused by choosing an easy (*limited*) solution now instead of using a better approach that would take longer.

Software systems are prone to the buildup of **cruft** (*badly designed, unnecessarily complicated*) - deficiencies in internal quality that make it harder than it would ideally be to modify and extend the system further. Technical Debt is a metaphor, coined by Ward Cunningham, that frames how to think about dealing with this cruft, thinking of it like a financial debt. The extra effort that it takes to add new features is the interest paid on the debt.



## Common Causes of Technical Debt

Common causes of technical debt include:

* Insufficient up-front definition, where requirements are still being defined during development, development starts before any design takes place. This is done to save time but often has to be reworked later.
* Business pressures, where the business considers getting something released sooner before all of the necessary changes are complete, builds up technical debt comprising those uncompleted changes.
* Lack of process or understanding, where businesses are blind to the concept of technical debt, and make decisions without considering the implications.
* Tightly-coupled components, where functions are not modular, the software is not flexible enough to adapt to changes in business needs.
* Lack of a test suite, which encourages quick and risky band-aids to fix bugs.
* Lack of documentation, where code is created without necessary supporting documentation. The work to create any supporting documentation represents a debt that must be paid. [[6]](https://en.wikipedia.org/wiki/Technical_debt#cite_note-SuryanarayanaSamarthyam2014-6)
* Lack of collaboration, where knowledge isn't shared around the organization and business efficiency suffers, or junior developers are not properly mentored.
* Parallel development on two or more branches accrues technical debt because of the work required to merge the changes into a single source base. The more changes that are done in isolation, the more debt is piled up.
* Delayed refactoring – As the requirements for a project evolve, it may become clear that parts of the code have become inefficient or difficult to edit and must be refactored in order to support future requirements. The longer that refactoring is delayed, and the more code is added, the bigger the debt.
* Lack of alignment to standards, where industry standard features, frameworks, technologies are ignored. Eventually integration with standards will come, and doing so sooner will cost less (similar to 'delayed refactoring').
* Lack of knowledge, when the developer simply doesn't know how to write elegant code. [[7]](https://en.wikipedia.org/wiki/Technical_debt#cite_note-Sterling2010-7)
* Lack of ownership, when outsourced software efforts result in in-house engineering being required to refactor or rewrite outsourced code.
* Poor technological leadership, where poorly thought out commands handed down the chain of command increase the technical debt rather than reduce it.
* Last minute specification changes, these have potential to percolate throughout a project but no time or budget to see them through with documentation and checks.

## Technical Debt Quadrant

"Technical Debt Quadrant", Martin Fowler distinguishes four debt types based on two dichotomous categories: the first category is reckless vs. prudent, the second, deliberate vs. inadvertent.

|  |  |  |
| --- | --- | --- |
|  | **Reckless** | **Prudent** |
| **Deliberate** | "We don't have time for design" | "We must ship now and deal with consequences (later)" |
| **Inadvertent** | "What's Layering?" | "Now we know how we should have done it" |