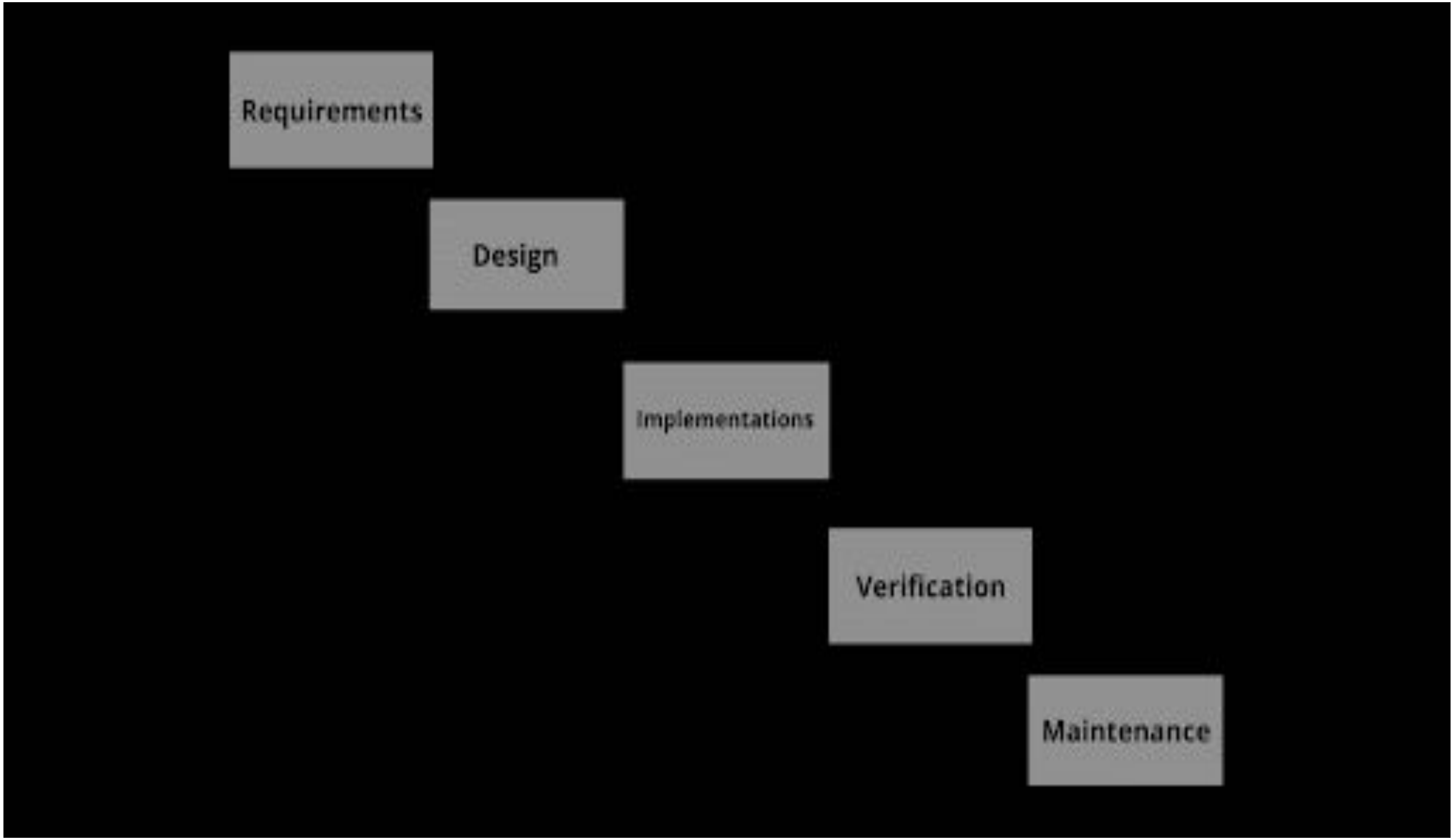
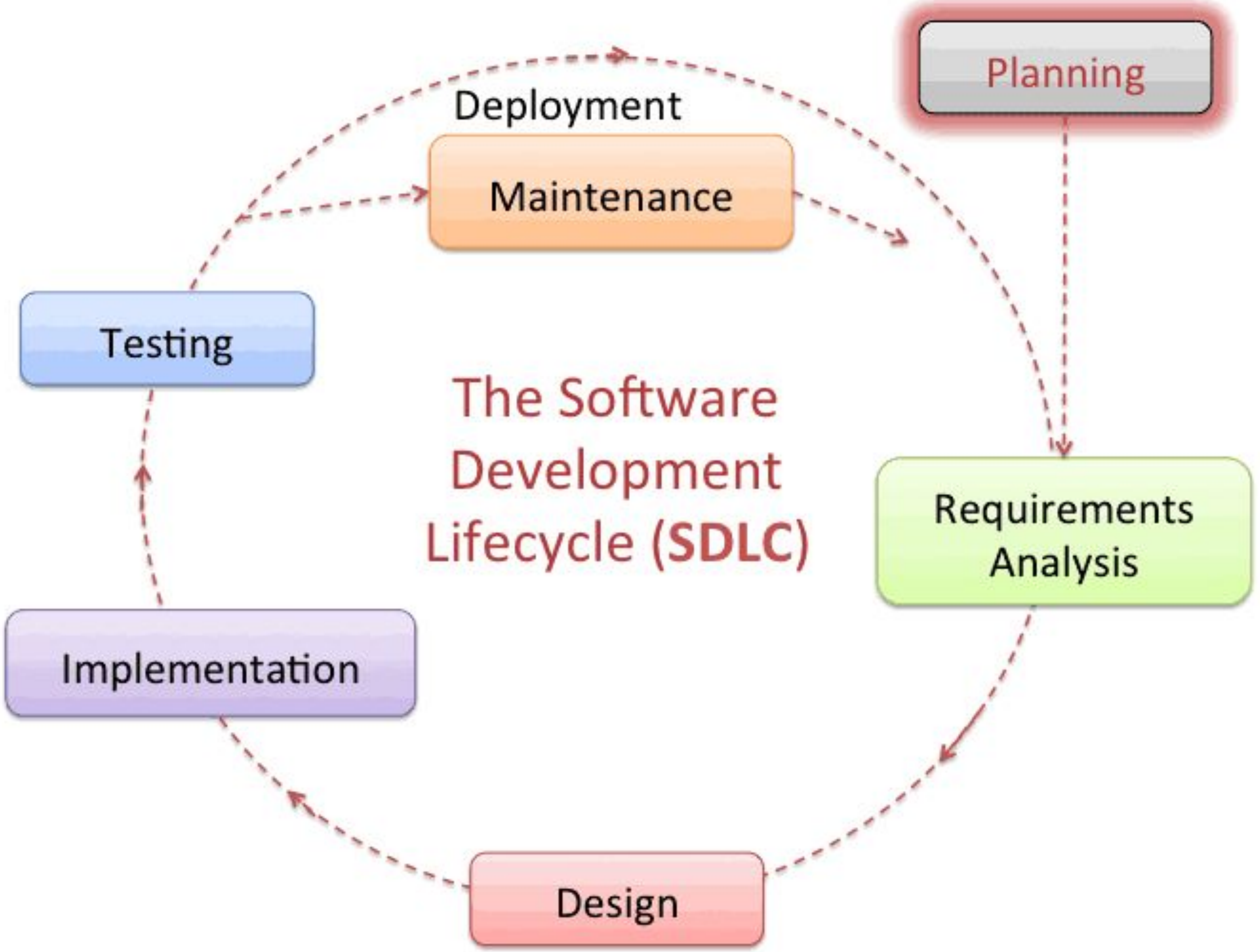
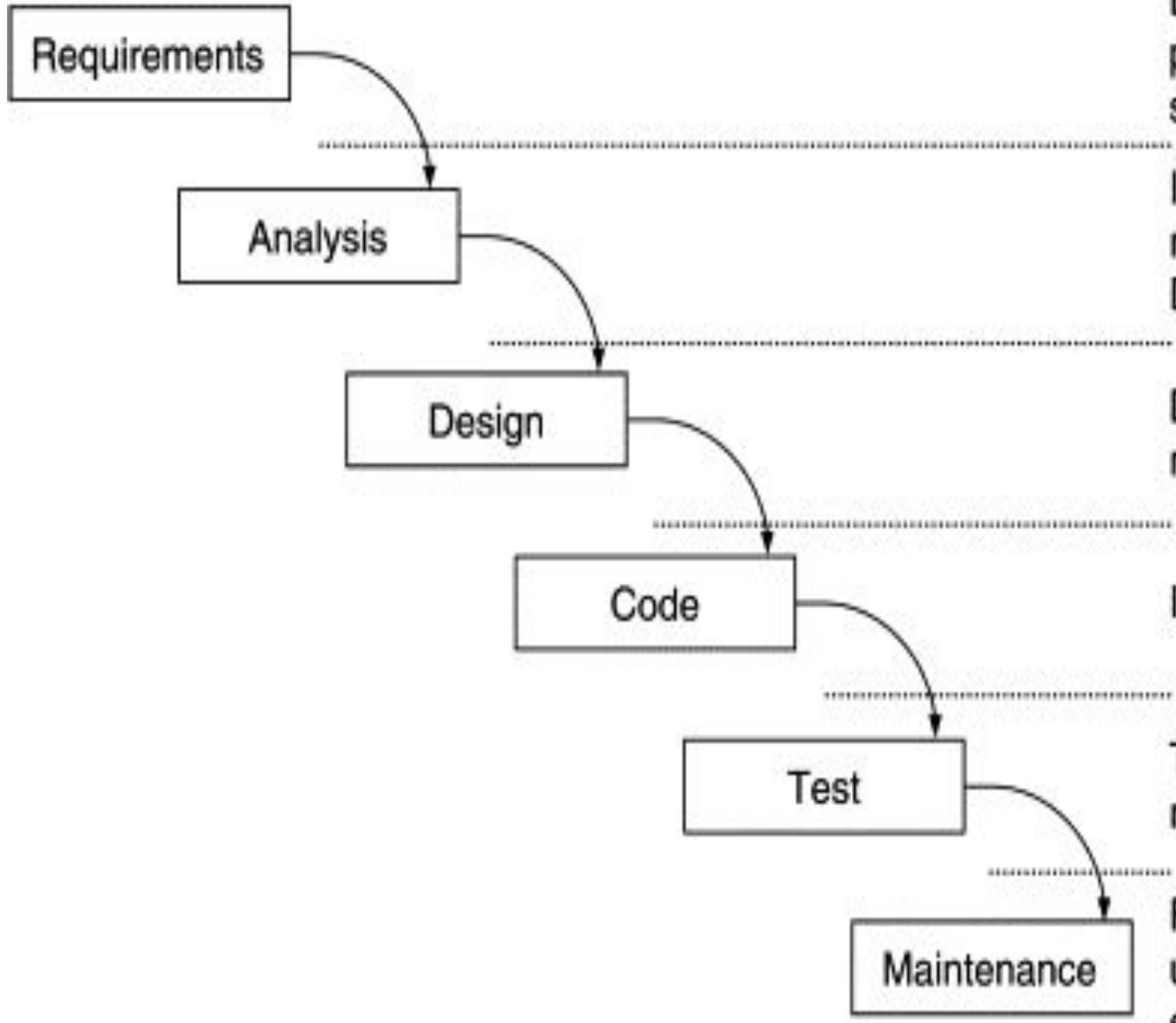
Life cycle of a software system


Requirements
Design
Implementations
Verification
Maintenance


Planning
Deployment
Maintenance
Testing
The Software Development Lifecycle (**SDLC**)
Requirements Analysis
Implementation
Design



Define the problem. Identify scope of problem and develop plan/strategy to solve the problem.

Investigate problem to define requirements necessary for solving it. Define "what."

Design a solution based upon the requirements. Define "how."

Implement (program) the design.

Test the program to ensure the requirements have been satisfied.

Fix or improve applications based upon use or changes in the environment.

- **Initiation**
  - An initiator **comes up with the initial idea**.
  - **Initiator** can be a **customer, executive champion, or software manager**.

- **Concept development**

- The initiator **explores the concept** to see if it's worthwhile and to **evaluate possible alternatives.**
- This **step** includes an **initial project definition, a feasibility analysis, a cost‑benefit analysis, and a risk analysis.**
- After concept development, **you should have enough information to make an informed decision.**
- The decision made at this point is a big one and it gets harder and more expensive to cancel the project as it starts to staff up and gain momentum.

- **Preliminary planning**

- A **project manager (PM) and technical lead are assigned to the project**, and they start planning.
- If it's a **big project**, the project might be **broken into teams and team leads would be assigned.**
- All these **leaders make preliminary plans** to **estimate** necessary resources such as **staffing, computers, network, development tools etc**
- The leaders **gather the tools needed to track and manage the project**.
- The **technical managers** also **decide on the development model, programming language, development environment, coding tools, and code conventions.**

- **Requirements analysis**
- The team **studies the user's needs and creates requirement documents**.
- Those may **include text, pictures, use cases, prototypes, and long-winded descriptions of business rules**.
- It may also include **UML diagrams showing application structure, user behavior, and anything else that helps the users understand** what the team will be building.
- The team also builds **technical requirements that let the developers know what they need to build.**

Characteristics of good requirements

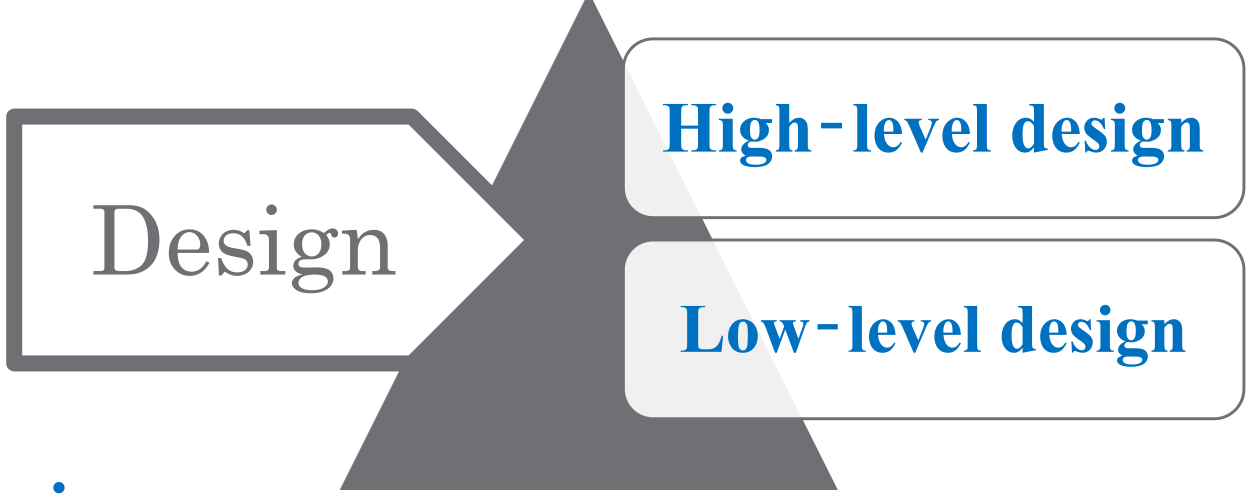
- **Clear**
  - Good requirements are clear, concise, and easy to understand.
  - Each requirement must state in concrete

- **Unambiguous**
  - Requirement must be written carefully to make sure that it is never interpreted them other than the way its intended.

- **Consistent**
  - A project’s requirements must be consistent with each other.
  - That means not only that they cannot contradict each other but also be self‑consistent.

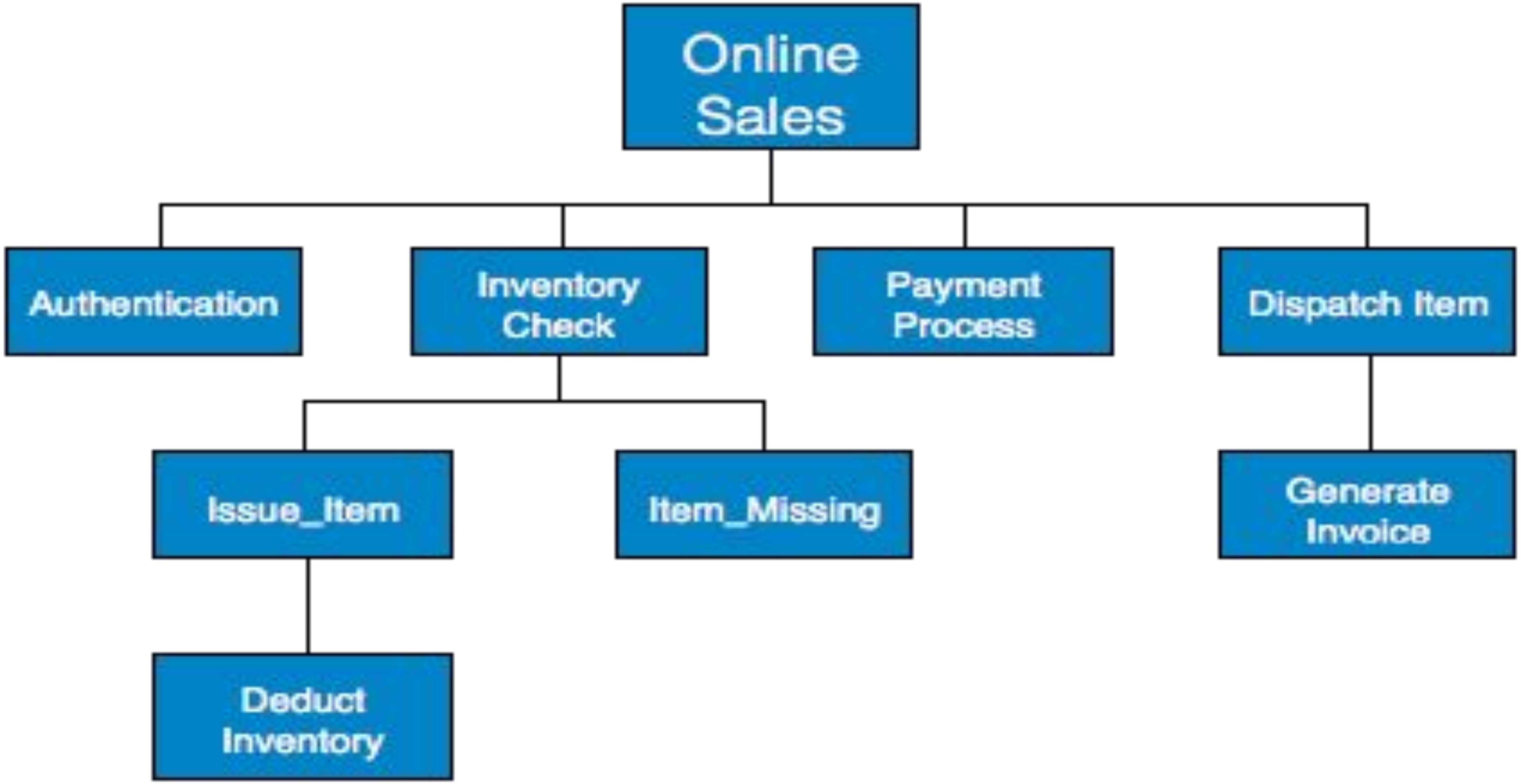
- **Prioritized**
  - If we include every feature but don’t have the time or budget, then we need to prioritize the requirements.
  - If we have assigned costs (eg:time to implement) and priorities to the requirements, then we can defer the high‑cost, low‑priority requirements.

- **Verifiable**
  - Requirements must be verifiable.
  - Being verifiable means the requirements must be limited and precisely defined.



- **High-level design**
  - The team creates high-level designs that specify **major subsystems, data flow, database needs, and the rest of the application's high-level structure**.
  - The high-level design includes things such as decisions about **what platform to use** (such as desktop, laptop, tablet, or phone), **what data design to use**.
  - The high-level design should also include information about the **project architecture** at a relatively high level.
  - We break the project into different modules that handle the project's major areas of functionality.

Example-HIPO Diagram



Low-level design

- The team creates low-level designs that **explain how to build the application's pieces.**
- After high-level design breaks the project into pieces, we can assign those pieces to groups within the project so that they can work on low-level designs.
- The low-level **design includes information about how that piece of the project should work.**
- **Better interactions between the different pieces of the project that may require changes here and there.**

- **Development**
  - The team **writes the program code**.
  - They **follow good programming practices**.
  - They **perform unit tests, regression tests, and system tests**.
  - They **fix the bugs that inevitably appear and handle any change requests that are approved by the change committee.**
  - The team **also prepares user documentation and training materials**.

- **Testing**
  - Even if a particular piece of code is thoroughly tested and contains no (or few) bugs, there's no guarantee that it will work properly with the other parts of the system.
  - One way to address the problems like this, is to **perform different kinds of tests**.
  - **First developers** test their own code. **Then testers** who didn't write the code test it.
  - After a piece of code seems to work properly, it is integrated into the rest of the project, and the **whole thing is tested to see if the new code broke anything.**

- **Deployment**
  - The software is **delivered to the customer** who evaluates the delivered product and provides **feedback** based on the evaluation.

- **Maintenance**
  - As the user starts using the software they would find bugs in them and **when the users find bugs, we need to fix them**
  - The team continues to track the application's usefulness throughout its lifetime to determine whether it needs repair, enhancement, or replacement with a new version or with something completely different.
  - If our application is successful, users will use it a lot, and they'll be even more likely to find bugs. They also think about of **enhancements, improvements, and new features** that they want to add immediately.

Thank You