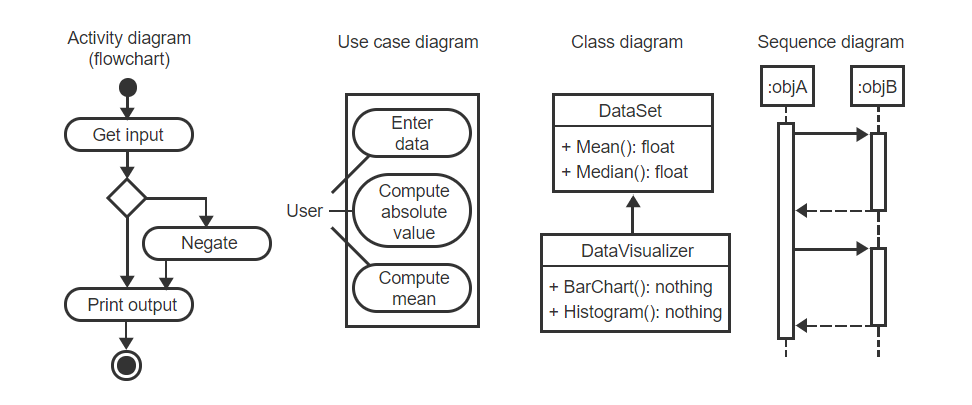
**UML | Unified Modeling Language**

A modeling language used to visualize requirements and design software. Used in the planning process prior to when code is written.

For a walk-through of these concepts, watch the [UML and the Software Development Life Cycle webinar](https://wgu.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=09eaffdc-95f7-45e6-a890-a87a01553236) (15 min)

Check your understanding of the concepts with the [C173/D278 Study Questions](https://srm--c.na127.visual.force.com/apex/coursearticle?Id=kA00c0000010yIjCAI).

**UML Diagrams**

**Use Case** - describes a singular goal of one user and briefly outlines how they will accomplish the goal (watch this video: [use cases](https://www.linkedin.com/learning/programming-foundations-object-oriented-design-3/use-cases), 5min)

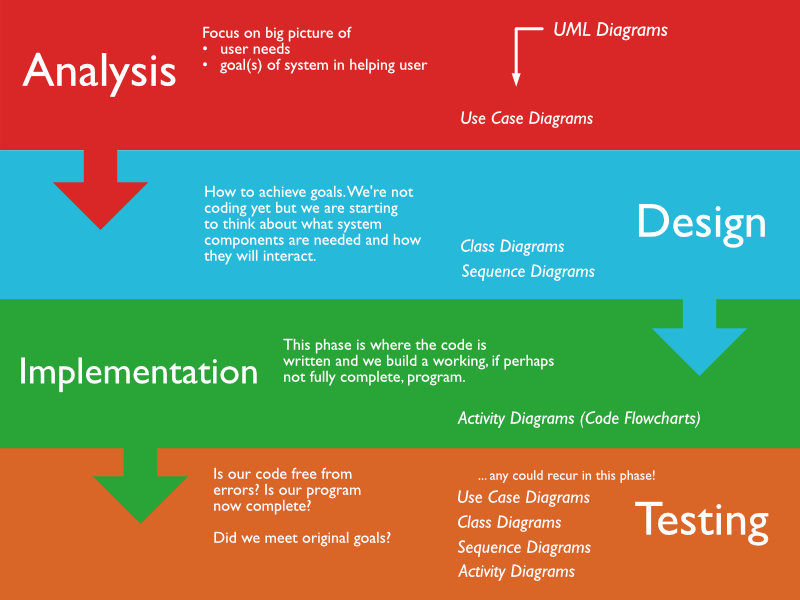
**Use Case Diagram** - models how a user interacts with a program (watch this video: [diagramming use cases](https://www.linkedin.com/learning/programming-foundations-object-oriented-design-3/diagramming-use-cases), 3min)

**Class Diagram** - models the classes, or objects of a program (watch these videos: [Creating class diagrams: Attributes](https://www.linkedin.com/learning/programming-foundations-object-oriented-design-3/creating-class-diagrams-attributes), 2min & [Creating class diagrams: Behaviors](https://www.linkedin.com/learning/programming-foundations-object-oriented-design-3/creating-class-diagrams-behaviors), 3min)

**Sequence Diagram** - interaction between software components and order of events

**Activity diagram** - a flowchart of an activity (loop, function, etc.) within the program

**Software Design Process**



Testing phase may look at any previous diagrams to ensure that the program was designed as intended. The purpose at this phase is no longer design before coding, but testing after coding.

**Waterfall vs Agile**

|  |  |
| --- | --- |
| **Waterfall** | **Agile** |
| Waterfall works in linear manner, no earlier phase is revisited. | Agile works in circular manner, each phase is revisited several times. |
| Pros:   * Simple and easy to understand and use * Easy to manage due to the rigidity of the model. Each phase has specific deliverables and a review process. * Phases are processed and completed one at a time. * Works well for smaller projects where requirements are very well understood. * Clearly defined stages. * Well understood milestones. * Easy to arrange tasks. * Process and results are well documented. | Pros:   * Is a very realistic approach to software development. * Promotes teamwork and cross training. * Functionality can be developed rapidly and demonstrated. * Resource requirements are minimum. * Suitable for fixed or changing requirements * Delivers early partial working solutions. * Good model for environments that change steadily. * Minimal rules, documentation easily employed. * Enables concurrent development and delivery within an overall planned context. * Little or no planning required. * Easy to manage. * Gives flexibility to developers. |
| Cons:   * No working software is produced until late during the life cycle. * High amounts of risk and uncertainty. * Not a good model for complex and object-oriented projects. * Poor model for long and ongoing projects. * Not suitable for the projects where requirements are at a moderate to high risk of changing. So, risk and uncertainty is high with this process model. * It is difficult to measure progress within stages. * Cannot accommodate changing requirements. * Adjusting scope during the life cycle can end a project. * Integration is done as a "big-bang. at the very end, which doesn't allow identifying any technological or business bottleneck or challenges early. | Cons:   * Not suitable for handling complex dependencies. * More risk of sustainability, maintainability and extensibility. * An overall plan, an agile leader and agile PM practice is a must without which it will not work. * Strict delivery management dictates the scope, functionality to be delivered, and adjustments to meet the deadlines. * Depends heavily on customer interaction, so if customer is not clear, team can be driven in the wrong direction. * There is a very high individual dependency, since there is minimum documentation generated. * Transfer of technology to new team members may be quite challenging due to lack of documentation. |