**Topic 1:**

**Types of Software:**

* Custom: for a specific customer, usually developed in house.
* Generic: sold on the open market, AKA, COTS (Commercial Off The Shelf) and shrink wrap.
* Embedded: built into hardware, hard to change.
* Real time software vs. Data processing software

Software engineering: The process of solving customers’ problems by the systematic development and evolution of large, high quality software systems within cost, time and other constraints.

**Software external qualities:**

* Usability: Users can learn it faster and get their job done easily.
* Efficiency: It doesn’t waste resources such as CPU time and memory.
* Reliability: It does what it is required to do without failing.
* Maintainability: It can be easily changed.
* Reusability: Parts can be used elsewhere; no need to reinvent the wheel.

Software quality from a stakeholder point of view:

* User: usable, efficient, and reliable.
* Customer: reliable.
* Developer: maintainable, reusable.
* Development manager: usable, efficient, reliable, and reusable.

**Software internal qualities:** The amount of commenting, the complexity, and the coupling of the code.

**Software Engineering Projects**

* Evolutionary (maintenance):Corrective projects, Adaptive projects, Enhancement projects, Reengineering or perfective projects.
* Greenfield: new project, a minority.
* Projects that involves building on a framework/ set of existing components:
  + The framework is an application missing some important details

**Activities Common to Software Projects**

* Requirements and specification: define problem, requirements gathering, analyzing and specifications.
* Design: decide how to implement the project using available technology.
* Modeling
* Programming
* Quality assurance: testing.
* Managing the process

**Difficulties/Risks in Software Engineering:** Complexity of the program, uncertainty about technology, requirements and software engineering skills, constant change, and deterioration of software design.

**Topic 2:**

Project management: encompasses all the activities needed to plan and execute a project.

**Software process models**: general approach to organize the project into activates.

* The opportunistic approach: bad approach. First prototype {while not satisfied do improve.}
  + Does not work out the requirements and the design before the implementation.
  + Does not have quality assurance.
* **The waterfall model**: accounts for requirements, design and quality assurance.
  + Requirements <-> Specs <-> Design <-> Implementation <-> integration <-> maintenance
  + Implies that some steps have to be done before others, assures the quality of each step and tracks back if there are any problem in the previous steps.
  + A part pf the Big Design Up Front (**BDUF**) approaches family.
  + Problems:
    - The previous stages must be complete.
    - Cannot be used until the entire system is complete.(No prototypes)
    - Implies that you get the requirements right, right of the pat.
    - Assumes that once the product is finished everything else is maintenance.
* **The Spiral Model**: adds prototyping and iterative approaches to SE.
  + Prototype -> LOOP -> Release #i -> Analysis of risk -> waterfall -> review -> i++
  + Maintenance is a type of ongoing development.

Agile approaches: development with smaller iterations. Stakeholders work closely, user stories, small and frequent releases, test-driven development (**TDD**), a lot of refactoring, pair programming.

**Re-engineering:** clean-up code, refactor code and replacing a code layer. Increases maintainability.

Principles of cost estimations:

* Divide and conquer: divide into really small tasks, give each task an estimation. Sum them up.
* Include all activities: prototyping, design, inspection, testing, debugging, documenting, etc.
* Base estimates on past experience
* Account differences: developers, software, customers, technology, complexity and stability.
* Anticipate the worst case and plan contingencies. **O**ptimistic-**L**ikely-**P**essimistic.
* Combine multiple independent estimates.
* Revise estimates as you progress.

Building teams:

* **Egoless**: everyone is equal, everyone contributes, better for difficult projects.
* **Hierarchy**: everyone reports to their manager, everyone has a specific task.
* **Chief Programmer**: the chief guides the projects, consults with the team specialists.

Brooks’ Law: adding manpower to a late projects delays it.

**Difficulties/Risks in project management**: difficult to accurately estimating costs, measure progress, meeting deadlines, dealing with the lack of human resources or technology, communication and obtaining agreements and commitments from others.

**Topic 3:**

Object: a chunk of structured data in a running software system, has properties and behaviors.

Class: A unite of abstraction in an OO program. Instances = properties and methods = behaviors.

**Class vs Object:** a class has instances, an instance is an object if it’s a single member of a class.

Naming class: use capital letter, singular nouns, right level or generality, only has one meaning.

**Polymorphism**: overloading methods.

Determine sub-classes using the “**is-a”** rule.

**Concepts that define an Object:**

* Identity: each object is distant from the other, even if the two objects have the same data.
* Classes
* Inheritance: subclasses inherit behavior from there super-classes.
* Polymorphism: overloading.

OO allows encapsulations, hiding information, ex private.

**Programming style:**

* Adhere to a good OO principle, such as the “is-a” rule.
* Prefer private as opposed to public.
* Prefer composition over inheritance.
* Don’t mix UI code with non-UI code.

**Difficulties/Risks in OOP:** language evolution and deprecated features, efficiency can be a concern in some OO systems.

**Topic 4:**

BDUF Problems:

* Difficult to envision every possible feature.
* Process of documenting the requirement is tedious and error-prone.
* Requirements are long and boring.
* Time wasted writing the requirements and deciding which one can be implemented in time.
* Levels of communication between customers and developers.
* The requirements might need lead to a good understanding of the product.
* Customers change their mind as the software is being built.

**User Stories**: describe functionality valuable to a user/customer.

* **Card**: a written description of the story.
  + Works as a reminder to have conversation.
  + Front of Card: User story, estimate points.
  + Back of Card: The expectations of the user story, and conformation.
* **Conversation**: conversations about the story to flesh out the details.
* **Confirmation**: tests that convey the document details.
  + Captures the important aspects of the story.
  + A set of tests to ensure that these aspects are met.

User stories must be written in the language of business by a customer team.

BDUF vs. Story-Driven Development

* In BDUF customers are only needed at the beginning of the project and at the end, while in SDD the customers are involved throughout the project.

**Writing good stories INVEST**

* **I**ndependent: Avoid dependencies
* **N**egotiable: The story should not have too much detail.
* **V**aluable: Avoid stories valued to developers, focus on stories valued to both sides.
* **E**stimable: lack of domain and technical knowledge, have a big story are reasons of failure.
* **S**mall: not too small, makes the job harder.
* **T**estable: try to have 99% test automation.

**Topic 5:**

Unified Modeling Language (**UML**)

**UML Diagrams**

* Class diagrams
* Interaction diagrams: show how objects interact with each other.
* State and activity diagrams: show how systems behave internally.
* Component and deployment diagrams: show how components are arranged.

**UML Components**

* Classes: represents the types of data.
* Associations: linkage between classes.
* Operations: methods.
* Generalizations: group classes into inheritance hierarchies.

Avoid unnecessary one to one associations.

**Aggregation**: refers to ownership.

**Composition**: a stronger aggregation, if the aggregate dies, so will the aggregated components.

**Interface**: is a class that does not contain instances and implemented methods.

* Reduce coupling.
* Rule: can-be-seen-as

UML Models:

* Exploratory domain model: developed in domain analysis to learn about the domain.
* System domain model: models aspects of the represented by the system.
* System model: Includes also classes used to build the user interface and system architecture.

**Topic 6:**

Integration Hell may lead to exceed the project schedule or budget.

Continuous Integration **CI**

Benefits of a CI Server

* Automatically runs tests are returns the results.
* It might take long to run tests.
* Variety of environments.
* Provides useful reports on the code.

Jenkins on Amazon Elastic Compute Cloud **EC2**.

**Topic 7**

Utility: the raw capabilities to allow the user to achieve their goal.

Usability: allowing the user to learn and to use the raw capabilities easily.

* Learnability: the speed at which a new user can become a proficient with the system.
* Efficiency of use: how fast an expert can do their work.
* Error handling: he extent at which it prevents the user from making errors.
* Acceptability: the extent to which the users like the system.

Usability Principle:

* Do not rely only on usability guidelines – always test with users.
* Base UI designs on users’ tasks.
* Ensure that the sequences of actions to achieve a task are as simple as possible.
* Ensure that the user always knows what they can or should do next.
* Provide good feedback including effective error messages
* Ensure that the user can always get out, go back or undo an action.
* Ensure that the response time is adequate.
* Use understandable encoding techniques.
* Ensure that the UI’s appearance is uncluttered.
* Consider the needs of different groups of users.
* Provide all necessary help.
* Be consistent.

Heuristic and user observation evaluations

**Difficulties/Risks in UI Design**

* Users differ widely
* User interface implementation technology changes rapidly
* User interface design and implementation can often take the majority of work in an application.
* Developers often underestimate the weaknesses of a GUI

Textual User Interface TUI

* Provides reasonable defaults.
* Provide short and long options.
* Provide standard options.
* Running a command with no arguments should display its usage details.
* Break up complex commands into sub-commands.
* Provide a man page so that users can RTFM!

**Topic 8**

**Architecting and Design Principles**

* **Divide and conquer**
* **Increase cohesion**
  + Functional: sort the code by its functionality.
  + Layer: sort the code into a hierarchy of layers, higher layers can access lower layers, while lower layers can’t access the higher ones.
  + Communicational: any classes or methods that communicate with each other is kept together.
  + Sequential: sorts the methods into steps.
  + Procedural: Keep together all the procedures that are used after each other.
  + Temporal: operations that are executed during the same phase are kept together.
  + Utility: when related to utilities and does not fit into other cohesive units.
* **Reduce Coupling**
  + **Content**: when a component modifies data that is internal to another component, instance variables that are not declared private.
  + **Common**: when using global variables.
  + **Control**: when a procedure calls another using a flag or a command that controls what procedure does.
  + **Stamp**: when passing an object as an argument.
  + **Data**: whenever you pass primitive types as arguments.
  + **Routine Call**: when a method calls another.
  + **Type use**: when using an object as an instance.
  + **Inclusion/Import**: when importing packages. Use wildcard imports.
  + **External**: when the module depends on the OS.
* **Keep a high level of abstraction.**
* **Increase reusability**
* **Reuse existing designs and code**
* **Design for flexibility**
* **Anticipate obsolescence**
* **Design for portability**
* **Design testability**
* **Design defensively**

Low/loose coupling is good, high/tight coupling is bad.

**Topic 9**

**Object Oriented Design Principles**

* Encapsulate what varies.
* Code to an Interface not an implementation.
* Favor Composition over Inheritance.
* **S**ingle Responsibility Principle.
* **O**pen/Closed Principle: open for extension, closed for modification.
* **L**iskov Substitution Principle (**LSP**): checks if a sub-class can replace the super-class.
* **I**nterface Segregation Principle (**ISP**): It makes sure that no interface implements another if it doesn’t need to.
* **D**ependency Inversion Principle: gives more freedom by replacing the implementations with interfaces.
* Principle of least Knowledge: a code component should not have high coupling instead the relation between its instances and the information should be direct.
* Don’t Repeat Yourself Principle (**DRY**): don’t copy and modify.

**Topic 10**

Test case: a set of inputs and outputs that are used to test the correctness of the program.

Test Suite: a set of test cases.

**Levels of testing:**

* Unit testing: testing individual methods and classes.
* Integration testing.
* System testing.
  + End-to-end testing.
  + Acceptance testing.

Methods of software testing:

* White-Box Testing: finished code testing.
  + JUnit
* Black-Box Testing: tests the requirements.

**Stubs**: a function that stands in for another function, that is unfinished.

xUnit testing phases: (State verification)

* Setup
* Exercise
* Verify
* Teardown

**Mocks**: objects the mimic the behavior of real objects.

**Topic 11**

**Levels of code coverage:**

* C0 – Statement/line Coverage
* C1 – Branch Coverage
* C2 – Condition Coverage
* C3 – Multiple Condition Coverage
* C4 – Path Coverage

Line Coverage -> Node Coverage -> Edge Coverage -> Condition Coverage -> Multiple Condition Coverage -> Path Coverage

**Topic 12**

**Functional testing**: test based on the stated and unstated requirements.

Tests

* Possible outputs
* Valid and invalid inputs
* Around boundaries
* Extreme values
* Input syntax
* Guess at possible errors