1 Software maintenance

Software maintenance is the modification of a software product after delivery to correct faults, to improve performance or other attributes. It includes:

- Fixing coding errors
- Fixing design problems
- Adding additional requirements

Up to 80% of software development can be considered maintenance.

1.1 Types

- Corrective
- Adaptive
- Perfective/performance Maintenance

1.2 Challenges

- Understanding the client what do you need to do?
- Understanding the code where do you need to do it?
- Refactoring the code changing existing code
- Extending the code adding to existing code
- Working as a team how to actually do the work?
- Managing client expectations
- Managing maintenance process

2 Diagrams

2.1 Class diagram

Class diagram is the main building block of object-oriented modelling. It is used for **general conceptual modelling** of the systematic of the application, and for detailed modelling translating the models into programming code.The classes in a class diagram represent both the main elements, interactions in the application, and the classes to be programmed.

2.1.1 Why

- Can be mapped directly with object-oriented language.
- Helps to define complex systems.
- Is a programmatic view of the system.

2.2 Use case diagrams

Usually referred to as **behavior diagrams** used to describe a set of **actions** (use cases) that some system or systems (subject) **should or can** perform in collaboration with one or more external users of the system (actors).

2.2.1 Why

- Easy for stakeholders to understand
- Show relationships between and among the actors and the use cases.
- Define functional requirements.
- Describe the system functions from the perspective of system users and in a manner they understand.

2.3 Activity diagram

The basic usage of activity diagram is similar to other four UML diagrams. The specific usage is to model the control flow from one activity to another. This control flow does not include messages. Activity diagram is suitable for modeling the activity flow of the system.

2.3.1 Why

- Show activity flow from one object to another.
- Describe the functionality of each object in the system
- Show whether actors can perform cases together or independently from one another.
- Allow you to think functionally.

2.4 Sequence diagrams

Sequence diagrams are a **temporal** representation of objects and their interactions; they shows the objects and actors taking part in a collaboration at the top of dashed lines

2.4.1 Why

- These can help to predict how a system will behave
- Discover responsibilities a class may need to have.

2.5 State machine diagram

State machine diagram is a **behavior** diagram which shows discrete behavior of a part of designed system through finite state transitions.

3 Build scripts

A set of instructions for how the project should be compiled or built. As projects get bigger or more complex, or rely on more external resources, custom build scripts are needed.

- They can build in dependencies, package files, run tests, deploy software etc.
- They also help to prevent human error.
- Portable Configurable
- Can script other external processes Testing Metrics Deployment
- Ensures consistent compiles and less Classpath etc. problems

4 Patterns and design

SOLID principles:

- Single responsibility principle - changes to only one part of the software's specification should be able to affect the specification of the class
- Open/closed principle - Classes should be extendible, but not modifiable. That means that adding a new field to a class is fine, but changing existing things are not. Other components on the program may depend on said field.
- Liskov substitution principle - objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program. (Basically work with subclasses of a class)
- Interface segregation principle - Keep your interfaces the smallest you can. (Use iV2, iV1 rather than iV1V2)
- Dependency inversion principle - Objects should not instantiate their dependencies, but they should be passed to them.

4.1 Object model

- **Abstraction** Denotes the essential characteristics of an object that distinguish it from all other kinds of objects.
- Encapsulation: mechanism for restricting direct access to some of the object's components
- Modularity: property of a system that has been decomposed into a set of cohesive and loosely coupled modules (classes).
- Hierarchy: A ranking or ordering of abstractions. Classes at the same level of the hierarchy should be at the same level of abstraction

4.2 Object oriented concepts

- **Data Abstraction** hides all but the relevant data about an object in order to reduce complexity and increase efficiency.
- **Inheritance** defines implementation of a class in terms of another's
- **Polymorphism** present the same interface for differing underlying forms (data types).
- **Interfaces** describes the behavior or capabilities of a class without committing to a particular implementation of that class

4.3 Design patterns

- Adapter Pattern (Structural) This pattern involves a single class which is responsible to join functionalities of independent or incompatible interfaces.
- Factory Method Pattern (Creational). With the Factory Method pattern we create objects without exposing the creation logic to the client and refer to newly created object using a common interface.
- Observer Pattern (Behavioural). Observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically
- State Pattern (Behavioural) We create objects which represent various states and a context object whose behavior varies as its state object changes.

Reference section

placeholder