# SE - Quizzes and Questions (2023)

by LFMV

### Disclaimer

The following topics are not covered in this file since I couldn’t find enough references in the current course material (2023) or in general I think they are not relevant for the exam, still, it may be worth investigating more:

- Question about subversions? (questions in very old Exams < 2015)

- COCOMO (only 1 question found in past exams back to 2009 or something)

- Delphi estimation method.

- MISRA-C

### Processes

**What are the main differences between an iterative and a waterfall process?**

Waterfall only performs one iteration, waterfall is also a project financed with a fixed price model , agile instead is a time and material model.

**Describe shortly the Test Driven Development technique**

Write one test case that fails (using requirements), write corresponding code until test case passes, repeat until

all requirements are satisfied and all test cases pass

**What is the typical lifecycle for a change?**

1. Reception
2. Filter
3. Assessment/Ranking
4. Assignment
5. Implementation
6. Release

**When a project reuses external components, how the activities of requirements and design are impacted?**

Req: search and evaluate components, do gap analysis between what is required, what is offered by components

Design: integrate components in design (possibly using wrappers, adapters)

**In a maintenance process, what are the possible types of a change?**

Corrective, enhancement, evolutive

**Describe briefly the pair programming technique in Xtreme Programming**

Two people working on one machine to develop both production code and test cases. One person writes, the

other controls and suggests improvements / modifications. The pair reverses roles often.

**What is the core content of the ISO 12207 standard?**

Hierarchical List of activities in the software process, defines the main processes in software development, along with the entities responsible of them, and their products/activities.

ISO 12207 is an ISO standard used to manage the life cycle of a software based on 2 main principles: Modularity → Defining processes with the lower coupling and the higher cohesion possible. Responsibility → There is a responsible for every process. There are three type of processes: primary, supporting and organizational.

**Describe briefly the ‘Scrum’ agile process:**

Sprints iterations of fixed duration (max 4 weeks) producing a working application in increments.

Ranking of requirements by end user / customer (requirements backlog).

Stand up meeting (15’) every day for coordination.

**Describe the waterfall process, its pros and cons. (x2)** →

Activities (requirement, design, implementation, unit test, integration test, system test) are done in sequence (activity i+1 starts only after activity i is completed). Document oriented.

→ Pros: easy structure of activities; agreement on design allows to allocate tasks to many, distributed workers/companies.

→ Cons: delivery to customer and validation of requirements and system happen very late. Changes require to restart the process, slowness and lack of flexibility.

**Software processes. Describe the key points of the evolutionary process**

The evolutionary model solves the problem of re-doing the whole waterfall that is present in the Incremental model: it is similar to the Incremental model, but requirements can change at each iteration. It is similar to prototyping, since at each iteration an incremental prototype is produced and the last prototype developed is the final product.

**Highlight the key differences between the incremental and iterative software processes (x2)**

→ In incremental software process, requirement and design document are immutable like in the waterfall but the integration test is split and done incrementally, and each loop produce a part of the system delivered to customers for feedback. The iterative model consists in many iterations of the waterfall model, which of it produce a partial part of the project.

→ The difference from the Evolutionary is that at the first iteration not all the requirements are written, but also those are made incrementally, while Evolutionary have the requirements written at the first loop and only can modify it to adapt to user wills.

### Verification and Validation

**What are the key differences between testing and debugging? (x2)**

Testing tries to find failures, debug tries to discover and fix the correspondent fault(s).

**Describe shortly the perspective based inspection technique (x2)**

Inspection where readers use different points of view (ex end user, designer, tester) to read the requirement document.

**What are Function Points used for? How are they defined?**

Function points measure the size of an application system based on the functional view of the system. The size is determined by counting the number of inputs, outputs, queries, internal files and external files in the system and adjusting that total for the functional complexity of the system

**Describe the key steps in an inspection process of a requirement document (x3)**

Moderator choose participants and schedule event. He shows the goal of inspection and techniques to be used, for example perspective-based, scenario-based, defect-based reading Each participant read the document. Then they meet up to read the document together and discuss defects found. Defects have to be solved and then there is a follow up to verify fixes.

⚠ Inspection is a V&V technique where people read the code and search for defects. It's made by a group with the objective of finding defects with no correction. The people who read the code should not be the ones reviewing it as the author is emotionally involved to its program so it's brought to hide eventual defects.

**A defect can be injected, discovered, removed. Of course not injecting defects at all is the best option. Is this feasible? Argument briefly your answer.**

Answer: Not injecting defects is not feasible: software is not defect-free for definition, since it's the result of a human activity. Probability of inserting a defect writing or changing code is different from zero Adams's Law)

**What are the possible types of defects in a requirement document?**

Omission/ incompleteness

Incorrect Fact

Inconsistency/contradiction

Ambiguity

Extraneous Information

Overspecification (design)

Redundancy

What are the possible type of defects in a code module?

→ Syntax, type and semantic errors;

→ Use of undefined or non-initialized variables

→ Not used variable (data-flow)

🡪Symbolic execution (translation of mathematical formulas) → Bad smells (too long classes or list of attributes)

**Describe the difference between bottom up and top down integration testing:**

The bottom up incremental integration starts from the dependency graph. → First are tested the units that have no dependencies and then one unit at the time is integrated. If some defect is found, it should come from the last integrated unit or from the interaction, but not from the units already tested. → The top down incremental integration starts from the highest-level unit and it uses stubs, then integrates the units below one by one (that to be tested used stubs). This testing method fits well if we develop the software in a top down approach and can detect early architectural flaws. This saves the definition of drivers for the bottom classes but require writing stubs for them (the opposite of the bottom-up approach).

**Function A calls function B, that calls function C. You want to apply bottom up integration. How**

**do you proceed?**

Test C, test B+C, test A+B+C

**Function A calls function B, that calls function C. You want to apply top down integration. How do you proceed?**

Test A using a stub for B → Test A + B using a stub for C → Test ABC

**Describe the key concepts of ‘mutation testing’.**

A technique to evaluate how good a test suite is. Errors are inserted in the program under test

(mutations). The more mutations the test suite finds, the better the test suite.

**Describe what is an oracle in testing, and the key problems related to it:**

An **Oracle** is what the output of a test case is compared to. Oracles can be human or automatic:

* A **human oracle** is based on individual judgement, and on understanding of the requirements specification;
* An **automatic oracle** is generated from formal requirements specification, or is based from similar software developed from other parties, or from previous versions of the program (regression).
* Ideally automatically generated but in reality They are most of the times human-generated, and hence they are also subject to errors: even test cases can be wrong. In some cases, e.g. UI comparisons, the comparators cannot be automatic.

**In which cases an Oracle can be automatic?**

If previous, reliable version of software application is available, or if function can be expressed mathematically

**What measures can be used to evaluate the quality of software?**

- Number of defects found over a period of time (defect rate, MTBF)

- Number of defects / size

- User satisfaction (using questionnaires)

- (remark the question asked ‘measures’ not ‘properties’)

**What is the basic principle of the Visual approach to GUI testing? What is different with respect to**

layout-based scripted GUI testing?

Visual approach: graphic components (buttons, menus etc) are recognized via image recognition

Layout approach: graphic components are identified retrieving their ID (used by the graphic library) or by

specific unique properties.

**In the context of verification and validation, describe Weinberg’s law (x3)**

The creator of a program is unsuitable to test it – for emotional attachment to its code he does not want to find many errors and tends to overlook defects in it.

⚠

Remember also:

Howden Theorem → It's impossible to find an algorithm able to generate a finite ideal test (selected by a valid and reliable criterion)

ParetoZipf Law → 80% of defects come from 20% of modules: it's better to concentrate on the faulty ones

### Architecture and Design

1. Design Patterns: which leverage higher level mechanisms in describing the system.

Creational Patterns: Creational patterns describe techniques to manage the instantiation of new objects, with the aim of avoiding errors and creating too many low-level functions. Main creational patterns are detailed in the following.

**Describe Singleton pattern:**

Is a **creational pattern :** the context is a class representing a concept that requires a single instance. Without the adoption of the singleton pattern, the client could use that class in inappropriate ways, instancing many objects of that class. Using the singleton pattern, the client can use only one object of that given class. In the singleton pattern, the class itself is responsible for the control of its instantiation (that happens only once); an hidden private constructor ensures that the class can never be instantiated from the outside.

**Describe Abstract Factory pattern**

It is **creational pattern,** it is useful when an application must be developed for different operating systems. In the abstract factory pattern, a class is created dependently on which operating system the software is run, but the client does not see it: it only sees abstract classes, and does not know anything about which variant he is using or creating.

Structural patterns: describe techniques on how to organize classes and objects in order to form larger structures. They are divided in structural class patterns, which use inheritance to compose interfaces, and structural object patterns, which describe how to compose objects to realize new functionalities. In the following, the main structural patterns are described.

**Describe shortly the Facade Design pattern (x2)**

The facade is a wrapper having multiple public methods passing through him: it's a useful solution in order to enforce encapsulation and to prevent the illecit access of public meths by clients from outside.

**Describe shortly the Adapter Design pattern (x2)**

The adapter design pattern is a **structural** pattern used to provide an abstract solution to the interoperability problem between different interfaces. For example, speaking of barcodes, I could have different APIs provided by different manifacturers: every barcode has different function and different names so I can use adaptors to avoid changing everything.

**Describe Structural Patterns:**

Structural patterns describe techniques on how to organize classes and objects in order to form larger structures. They are divided in structural *class* patterns, which use inheritance to compose interfaces, and structural *object* patterns, which describe how to compose objects to realize new functionalities.

**Describe Adapter pattern:**

the adapter pattern is used when a class providing the required functionalities exists, but its interface is not compatible with the system. It converts the interface of the class, to match the expecting one, in order to integrate the class without modifying it: it proves useful especially when the source code of the class is not available, or the class is already used somewhere else as it is. Adapter pattern is often used with device drivers, a case in which a single interface for all the device drivers can be desirable, but individual original interfaces of the drivers may be not compatible with the required one.

In the Adapter pattern, the inheritance mechanism plays a fundamental role. It is the only example of structural class pattern.

An example of use of the Adapter pattern is the Java Listener Adapter: in Java, GUI events are handled by Listeners, and Listener classes need to implement Listener interfaces, with several methods that should all be implemented.

**Describe Composite Pattern:**

it is used in order to allow the user to logically group objects, by creating trees and represent part-whole hierarchies. Two types of objects are described: Composites and Leaves. They both share the same parent class. Composite objects, moreover, can group other objects and, occasionally, perform operations on them.

An example of usage of the Composite pattern can be a computer system, with a parent class, *Equipment*, deining the method watt(), used to compute the power consumption. Leaves objects can be *Hard Drive*, *Motherboard, CPU,* and so on; a Composite object, including leaves, can be *Server*. By applying the composite pattern, all the leaves can be added to the composite one, and by calling server.watt() the total power, for all the components, can be computed. Moreover, *serverFarm* can be defined, to which several servers can be added.

The composite pattern can be used also for the representation of Graphical User Interfaces, which expose to the user components that contain other components inside.

Behavioural Patterns: patterns describe the behavior of objects, the algorithms governing them and the responsibilities between them.

Behavioural patterns describe the behaviour of objects, the algorithms governing them and the responsibilities between them. They are more patterns of communication between objects, shifting away from complex control flows description to concentrate on the way objects are interconnected.

**Observer pattern:** it is used in a context in which changes in one object may influence one or more other objects, i.e. there is high coupling between objects. The number of type of objects influenced by changes in others may not be known in advance. The observer pattern creates an abstract coupling between subject and observer classes, that may be then instantiated by a variable number of concrete subjects and observers.

**Describe Template Method Pattern:**

it is used when an algorithm or behaviour has a stable core, and several variations at given points, and avoids the maintenance or implementation of several almost identical pieces of code. In the template method pattern, an Abstract Class contains the core algorithm, with abstract primitive operations; Concrete Classes implementing the Abstract Class define variants of the algorithm, without having to implement the primitive operations.

**Describe Strategy pattern:**

it is used when there are many algorithms that have a stable core and several behavioral variations, and it is desired to use one or another without changing the way to call it (since multiple conditional constructs tangle the code). In these cases, an interface for the algorithm is defined, and then many classes can implement it: this way, one or another class can be used without changing the caller (e.g., the Comparator in Java). There is a fundamental difference between Abstract Factory pattern and Strategy pattern: Abstract Factory describes the way objects are created, which involves the class itself with its own set of attributes and methods; Strategy pattern describes the way objects behave, while calling the same method.

Using the strategy pattern allows to avoid conditional statements and to organize algorithms in families, providing run-time binding of the proper algorithm to caller objects. The main drawbacks of the pattern are a communication overhead and an increased number of objects defined.

1. Architectural Patterns : which are ways of building system-wide structures

**Describe shortly the MVC Design pattern (x4)**

MVC is a software *ARCHITECTURAL* pattern commonly used for developing user interfaces that divides the related program logic into three interconnected elements called model, view and controller. It provides a separation of responsibilities at the cost of more complexity. ⚠ Model → The central component of the pattern. It is the application's dynamic data structure, independent of the user interface. It directly manages the data, logic and rules of the application. View → Any representation of information such as a chart, diagram or table. Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants. Controller → Accepts input and converts it to commands for the model or view

**Describe briefly the ‘repository’ architectural style:**

Several applications communicate only through exchange of (standardized) data files. No direct

interaction between applications, no call to APIs. Example: Eclipse.

(The question was about software architectures, not about repositories in context of configuration

management)

**Describe the ‘pipeline’ architectural style, and when it can be used:**

Many modules are connected in a sequence. Each module is independent of others. Data from one module flow

to the next one. Data is the only communication means to a module.

Ex: compiler linker. Ex: unix shell

**Describe Layered Arch Style:**

Also know as abstract machine, the idea of this pattern is to separate the software into layers or abstract machines, each one providing specific services.

The constraint posed on layers is that only functions of adjacent ones can be used. An example of a layered architecture is the ISO/OSI model for network architecture, organized in seven different layers, from Physical (pertaining wires and cables) to Application (web processes and applications).

From the architectural point of view, it forces the separation of concerns, since each layer is dedicated to solve one problem; moreover, the layered structure guarantees easy modifiability, since when a layer interface changes, only the adjacent layers are affected.

Beware that there are others architectural patterns but never appeared in past exams, I would suggest check them out in the SE HandBook: broker, client-side, micro kernel etc..

Other random question

**Given a set of functional requirements, often many designs are possible. How to select one design**

**option versus other ones?**

Several design can be generated to satisfy the FRs, but eventually Given that all designs satisfy the functional requirements, selection should be done considering NF, Requirements.

**What is a build? What are the related problems?**

Answer: Process (including compilation, link, possibly testing and other activities) to produce an executable starting from source code modules and libraries. Often automated with a script. Problems: references and dependencies among modules, finding and using the right modules in term of type and version.

**Provide an example of two conflicting non-functional properties of a software**

**Architecture.**

Very high precision of computation (ex square root precision 10-10) conflicts with Performance.

The higher the precision, the slower the response time.

Also security vs performance is a good example.

### Management

**What are the most important functions of a configuration management tool?**

Identify and manage part of software → Handle repository history → Handle branches, configurations and versions of software and eventually revert project back to a certain version (rollback) → Handle acc

**What is a Configuration?**

A configuration is a set of Configuration Items (CI), each in a specific version, and it can be seen

as a snapshot of the software at certain time. Some CIs may appear in different configurations,

and also configuration has a version number

**In the context of configuration management, what is the derivation history of a configuration**

**item?**

History of versions and changes to them

**In the context of project management, give the definition of ‘slack time’**

Admissible delay to complete an activity without changing the end time of project

**Check In/Check Out mechaninsm:**

Two main operations are performed on files that are extracted from the repository: **check-in** (commit), that updates the repository with the CIs local changes, and **check-out** (pull), that updates the workspace getting the CIs stored in the repository. These two basic operations are used to discipline the changes on the documents.

Nel Vecchio file dice:  
In the context of configuration management, what is the purpose of "check in" and "check out" operations? They are used to:

→ Enforce sequential changes to the Continuous Integration CI (in lock-modify-unlock)

→ Support parallel changes without inconsistencies (in push-modify-merge mode)

**Lock Modify Unlock strategy:**

is like a serialization of the changes. A developer tries to win a lock over the CI using the check-out operation; if nobody already has the lock on the CI, then it can be changed by the developer who requested the lock; otherwise, the developer cannot checkout the CI and must wait for lock release. The lock is released only when the developers holding it checks in.

The main problem of the Lock Modify Unlock approach is that if the locker forgets to unlock a CI, no other developer can modify it. Also, there is no possibility of parallel working: just a single developer can work on a CI. Hence, this approach can be too rigid for large projects and large teams of developers.

**Copy Modify Merge strategy:**

The **Copy Modify Merge** is a less rigid strategy, that allows many developers to check out the same file and then perform parallel work on it. The only issue is the necessity of confronting and merging the changes introduced by two or more developers on the same file, which can include conflicts (there are tools that perform automatic merge on modified CIs).

**What is a baseline?**

A **Baseline** is a special configuration that is in a stable form. Not all configurations are baselines. The baseline is often frozen in its state and deployed: modifications start by it, and if something erroneous occurs, the project is rollbacked to the baselyne. There are two types of baseline: a *development baseline* is for internal use, and is a safe rollback point for development; a *product baseline* is delivered to the user/customer, and is a stable version of the program.

**In the context of project management, give the definition of ‘deliverable’:**

A **Deliverable** is a product of the process. It may be either final or intermediate (e.g., a requirements document, a prototype, or the deployed application). A deliverable can be internal, if it must be used only in the producer company, or external, if it is done for the customer or must be validated by it. Some deliverables (like the requirements management or the design document) may have contractual values between customer and producer.

**In the context of project management, give the definition of ‘Milestone’:**

A **Milestone** is a key event or condition in the project, that can also serve as a synchronization point for its transitions between one phase to another. A milestone has effects on the subsequent activities: for instance, a milestone may be the acceptance (or refusal) of a requirement document by the customer. Whether it has been accepted or not, the activities to be performed later may have to change.

**Considering GIT, what are the three project sections that it defines, and how are they used?**

is a system used primarily for Source Code Management in software development, its main sections are:

1. Git directory, it stores the object database, along with the related metadata, for the whole project
2. Working directory, also called *Working Tree*, it is a checkout of a version of the project. It contains all the files as they are extracted from the compressed Git Folder, and that are stored locally to be used or modified by the developer.
3. Staging area: also called *Index*, it contains information about all the files that will be part of the next commit. Since the Data Management Model of Git is based on snapshots, snapshots of the files are added to the Staging are when they are staged.

**A software product is developed custom for a bank. Development takes 18 months,**

**operation and maintenance 10 years. Where do you expect to have the larger part of costs?**

Maintenance, because 10 years is way longer than 18 months, evolutive, corrective and

enhancement maintenance will likely require more effort than initial development.

**The size of a project is estimated to be 600 FP. The project is developed in Java, past**

**figures from the company tell that one FP is worth 20 to 30 LOC, and that productivity is 10-15**

**LOC per person days. How many person days would be required for the project?** (show steps

followed to compute your answer).

600 FP 12000 to 18000 LOC

Min effort 12.000 / 15 = 800 PD, max effort 18.000 / 10 = 1800 PD

**An application is developed for an organization in 6 months by 3 people. Then it is used by the**

**organization during 12 years. Argument whether maintenance costs could be higher /lower than development costs:**

Maintenance costs will probably be much higher than development cost, since the duration of operations (12

years) is way longer than development (6 months)

Effort for development: 3\*6 = 18 person months

Effort for maintenance: assuming 6 months per year (half a person maintaining the application) = 6\*12 = 72

person months

More precisely, if maintenance requires more than 18person months / 12 = 1,5 person months per year = 33

person days per year, then maintenance costs will be higher than development costs.

**The estimated effort for a new development project is 48 person months. Discuss what could be the estimated calendar duration of the project**

The rough formula is effort / #persons working full time If one person works full time on the project, 48 calendar months If two people work full time, 24 calendar months And so on. However, there is a limit to the max number of people that can work effectively in parallel, and therefore a limit to the min calendar duration. This limit can be computed by defining the Gantt chart for the project.