1 Elaborate on various levels of Abstraction with an example.

8.4.4 Design and Refactoring

Software design is carried out on a higher level of abstraction than code development. In other words, the development of a specific part of the software includes details which are eliminated when the design is constructed. Design is used to simplify communication with respect to the software and to represent the structure of the software components in an organized manner that can be understood by the different developers involved. Design can be sketched by some visual model and should be kept simple and clear.

Refactoring (Beck 2000, Fowler 1999, Highsmith 2002) or redesign means that we improve the software design without adding functionality. Refactoring is based on the current design and it attempts to simplify it and ease future changes. This activity requires thinking at a higher abstraction level than the level of abstraction needed when dealing with the design itself.

Reaching a simple design is not a simple task, and therefore refactoring is one of the practices that people find hard to accomplish. This difficulty can be

explained by the need to think about the code at a higher level of abstraction than the level of abstraction on which the code was written.

2 Why does a development process that is based on short iterations have many benefits?

The concept of short releases/iterations, and the planning sessions that direct the development process, encourage all the project stakeholders to move between levels of abstraction and to improve their understanding of the developed software gradually and periodically. While the release planning sessions inspire a global view of the developed product at a higher abstraction level, planning is conducted on a lower level of abstraction in the iteration planning sessions, considering the development tasks for the next iteration and their time estimation. The concept of short releases/iterations, and the planning sessions that direct the development process, encourage all the project stakeholders to move between levels of abstraction and to improve their understanding of the developed software gradually and periodically. While the release planning sessions inspire a global view of the developed product at a higher abstraction level, planning is conducted on a lower level of abstraction in the iteration planning sessions, considering the development tasks for the next iteration and their time estimation.

From the constructivist perspective, a development process that is based on short iterations has several benefits which are directly connected to learning processes.

First, it allows both the customer and the software team members to focus on a relatively small part of the iteration.

Second, it allows the customer and the software team members to gradually improve their understanding with respect to the developed software. This is because short iterations do not require dealing with future developments that are unknown at a specific stage, and that will probably be clarified later when the development proceeds.

Third, short iterations improves communication between among the project stakeholders in general and between the customer and the team in particular. The Business Day, which takes place after each short iteration, and in which the customer, the team, and management participate, enables all the project stakeholders to gather together, communicate, become familiar with the others' perspectives on the project, express their concerns with respect to the development process and the product, and reflect on previous developments. All these activities improve the understanding of the development process and the product by all the project stakeholders.

Fourth, short iterations define very clearly the time for feedback and reflective sessions. Feedback is provided by the customer at the end of each iteration; reflective sessions also take place at the end of each iteration.

Fifth, in addition to the lessons learned during such reflective sessions, such a break enables the developers to rest and detach for a while from the demanding, complex, and tight process of software development. It enables the team members to exploit their capabilities in a better way when they return to the development tasks of the next iteration.

Sixth, short iterations foster courage to raise problems and to attempt to solve them together. At the end of each iteration the team presents to the customer what has been accomplished during the last iteration, and if needed, shares with the customer any misunderstandings and/or problems in the development process.\

3 .Explain game theory with a case study

Game theory is concerned with the ways in which individuals make decisions, where such decisions are interdependent. For this purpose game theory uses theoretical fields such as mathematics, economics, and other social and behavioral sciences. The word ‘‘game’’ indicates the fact that game theory examines situa tions in which participants wish to maximize their profit by choosing particular courses of action, and in which each player’s final profit depends on the courses of action chosen by the other players as well.

The ‘‘prisoner’s dilemma’’ is a game theory framework that illustrates how a lack of trust leads people to compete with one another, even in situations in which they might gain more from cooperation. We will show that the transparency of agile software development eliminates the basic condition of the prisoner’s dilemma and thus increases **trust.**

The following analysis is based on Hazzan and Dubinsky (2005). In the simplest form of the prisoner’s dilemma, each of two players can choose, at every turn, between two actions: cooperation and competition. The working assumption is that none of the players knows how the other player will behave and that the players are unable to communicate.

Table 9.2 The prisoner’s dilemma in software teams (With kind permission of Springer Science and Business Media.)

	B cooperates	B competes
A cooperates	The project is completed on time. A and B get the bonus. Their personal contribution is evaluated as equal and they share the bonus equally: 50% each	A’s cooperation leads to the project’s completion on time and the team gets the bonus. However, since A dedicated part of his or her time to understanding the complex code written by B, while B continued working on his or her development tasks, A’s contribution to the project is evaluated as less than B’s. As a result, B gets 70% of the bonus and A gets only 30%
A competes	The analysis is similar to that presented in the cell “A cooperates/B competes.” In this case, however, the allocation is reversed: A gets 70% of the bonus and B gets 30%	Since both A and B exhibit competitive behavior, they do not complete the project on time, the project fails and they receive no bonus: 0% each

We now use the prisoner’s dilemma to analyze software development environ ments. It should be noted first that, in these environments, cooperation can be expressed in different ways, such as information sharing (or hiding), using (or ignoring) coding standards, clear and simple (or complex and tricky) code writing, etc. It is reasonable to assume that expressions of coopera tion will increase the project’s chances of success, while expressions of competition may add problems to the process of software development

Test-Driven Development.: The meaning of cooperation in the case of test driven development is that all team members verify that HOT their code is tested and does not fail; competition means that team members do not verify that the code is fully tested.

- practice of test-driven development yields a better outcome for all team members

Refactoring. : The meaning of cooperation in the case of refactoring is that all team members make sure to stop their development tasks from time HOT to time and improve their existing code readability and clarity (remember that time is allo cated in the planning sessions for big refactoring activities); competition means that the team members do not pay attention to code readability and improvement and do not invest the time needed for refactoring.

4 .Compare Quality with respect to agile and traditional approaches.

Answer :

Table 6.1 Some differences between agile and other methods with respect to quality

Quality-related aspect	The agile approach	Some other approaches
Who is responsible for software quality?	All the development team members	The QA team
When are quality-related topics addressed?	All of the time; quality is one of the primary concerns of the development process	At the QA/testing stage
Quality-related activities status	Same as other activities	Low (Cohen et al. 2004)
Work style	Collaboration with all parties	Developers and QA people might have conflicting interests

5 question : Define TDD process with a case study.

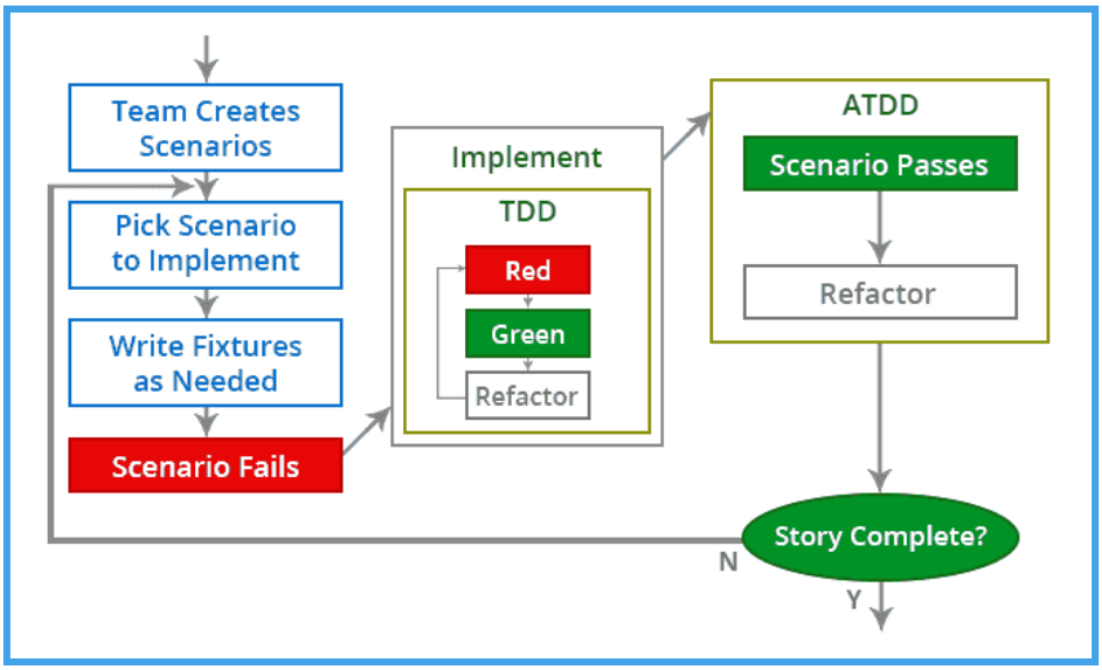
Answer :

(iv) Test Driven Development (TDD):

- TDD is a technique that enables a step-by-st development of a spefic functionality together with its unit test.
- It is a programming technique which aims to provide clean, fault-free code.
- First we write a test case that fails and then we write a possible code to pass the test successfully.

Test Driven Development (TDD) ***

- TDD is a programming technique which aims to provide clean and fault-free code.
- In TDD, we first write a test case that fails, and then we write a simple possible code to pass the testcase successfully.
- TDD implies that new code is added only if an automated test has failed.
- TDD guidline is
  - RED - writing a simple test that fails.
  - GREEN - write code that passes the test.
  - REFACTOR - code quality should be improved without adding/changing functionality.



TDD is a software-driven process that includes test-first development. It means that the developer first writes a fully automated test case before writing the production code to fulfill that test and refactoring. It is a practice of writing a (failing) test before writing the code of a feature. Feature code is refined until it passes the unit test.

Explain the above diagram with some example as a case study.