**1. Introduction to Refactoring**

**Definition**:

* Refactoring is the process of modifying software to improve its internal structure without changing its external behaviour.

**Key Points**:

* **Improves Design**: Makes the code easier to understand and cheaper to modify.
* **Helps Find Bugs**: By cleaning up the code, bugs can be more easily spotted.
* **Does Not Add New Functionality**: Focuses solely on improving the existing code structure.
* **Eases Addition of New Features**: A cleaner codebase makes it easier to add new functionality in the future.

**2. Why Refactor?**

**Key Reasons**:

1. **Improves Design**:
   * Prevents design decay due to changes over time.
   * Maintains the integrity of the original design, avoiding "design entropy."

**Example**:

* + Imagine a well-organized library. Over time, as books are added and removed, the organization may start to decay. Refactoring is like reorganizing the library to maintain its original order and ease of use.

1. **Makes Software Easier to Understand**:
   * Simplifies the structure, making it easier for developers to comprehend.
   * Reduces duplicated code and makes the system more maintainable.

**Example**:

* + In a cluttered workshop, finding tools is difficult. Refactoring is like reorganizing the workshop so tools are easy to find and use.

1. **Helps Find Bugs**:
   * Forces a deep understanding of the design, which can expose bugs.
   * Improves readability, making bugs easier to spot.

**Example**:

* + When cleaning a room, you might find things you didn't know were there. Similarly, refactoring can help uncover hidden bugs.

1. **Helps You Program Faster**:
   * Better design and readability lead to fewer bugs and faster development.
   * Although it may seem to slow down progress, it actually speeds up development in the long run.

**Example**:

* + Think of refactoring as regular maintenance on a car. It might take time initially, but it prevents bigger problems and repairs in the future, keeping the car running smoothly.

**Scenario**:

* **Client-Server-Based Enrolment System**:
  + Delivered in January.
  + Modified for a native mobile app in April.
  + Further modified to store enrolment data internally in September.
  + As requirements change, the system tends to drift away from its original design, introducing issues like sluggish performance and hard-to-maintain code.

**3. Refactoring Process and Techniques**

**Process**:

1. **Test-Refactor-Test**:
   * Ensure behaviour preservation through pre- and post-transformation testing.
   * Implement refactoring in small, manageable steps.

**Common Refactoring’s**:

* **Extract Method**: Create a new method from a code fragment.
  + **Example**: If a code block in a function is responsible for a distinct task, it can be moved to a separate method for clarity.
* **Move Method**: Move a method to a more appropriate class.
  + **Example**: If a method in Class A is using more features of Class B, it should be moved to Class B.
* **Rename Method**: Change a method name to better reflect its purpose.
  + **Example**: Change **calculate()** to **calculateTotalPrice()** to clearly indicate the method's function.
* **Pull Up Field**: Move a common field from subclasses to the superclass.
  + **Example**: If multiple subclasses have the same field, it can be moved to the superclass.
* **Push Down Method**: Move a method from a superclass to relevant subclasses.
  + **Example**: If a method in a superclass is only used by some subclasses, it should be moved to those subclasses.
* **Inline Method**: Replace a method call with the method's content to simplify the code.
  + **Example**: If a method simply returns a value or performs a very small task, it can be inlined for simplicity.

**4. Refactoring During Development**

**Agile Methods**:

* Encourages refactoring during initial development.
* **Two Hats**:
  + **Adding Functionality**: Write tests, add functionality, and ensure tests pass.
  + **Refactoring**: Improve code structure without changing its behaviour.

**Process**:

* **Add Functionality Hat**:
  + Write tests for new functionality.
  + Implement the functionality.
  + Ensure tests pass.
* **Refactoring Hat**:
  + Recognize when refactoring is needed while adding functionality.
  + Refactor the code to improve its structure.
  + Ensure behavior remains unchanged through testing.

**Example**:

* When adding a new feature, you might realize that the current code structure makes it difficult. By refactoring, you can reorganize the code to make adding the new feature easier.

**5. Examples of Refactoring**

1. **Rename Method**:
   * **Problem**: The method name does not reveal its purpose.
   * **Solution**: Change the method name to something more descriptive.
   * **Example**: Change **getCrLit()** to **getCreditLimit()**.
2. **Pull Up Field**:
   * **Problem**: Two subclasses have the same field.
   * **Solution**: Move the field to the superclass.
   * **Example**:

plaintext

Copy code

public class Employee { private String name; }

1. **Push Down Method**:
   * **Problem**: A method in a superclass is relevant only for some subclasses.
   * **Solution**: Move the method to those subclasses.
   * **Example**:

plaintext

Copy code

public class Manager extends Employee { void setIncrement() { // Method logic } }

1. **Move Method**:
   * **Problem**: A method is using more features of another class.
   * **Solution**: Move the method to the class it uses most.
   * **Example**:

plaintext

Copy code

class Order { double calculateTotalPrice() { // Calculate total price } }

1. **Inline Method**:
   * **Problem**: A method call is unnecessary.
   * **Solution**: Replace the method call with the method's content.
   * **Example**:

plaintext

Copy code

boolean hasDiscount(Order order) { return order.basePrice() > 1000; }

1. **Self-Encapsulate Field**:
   * **Problem**: Direct access to private fields inside a class is inflexible.
   * **Solution**: Use getters and setters for field access.
   * **Example**:

plaintext

Copy code

class Range { private int low, high; boolean includes(int arg) { return arg >= getLow() && arg <= getHigh(); } int getLow() { return low; } int getHigh() { return high; } }

**6. Refactoring Software Architecture**

**Architectural Refactoring**:

* Improves the design of existing software, changing the structure without altering functionality.
* **Goals**:
  + Improve performance, scalability, extensibility, testability, and robustness.
* **Example**:
  + Splitting a monolithic application into microservices to improve scalability and extensibility.

**Detailed Explanation**:

* **Monolithic Application**: A single, large application where all components are interconnected.
* **Microservices**: Breaking down the application into smaller, independent services that communicate with each other.
* **Benefits**: Each service can be developed, deployed, and scaled independently, improving overall system performance and flexibility.

**7. Controversy Around Refactoring**

**Research Findings**:

* Refactoring does not always result in higher quality code (measured by cohesion, coupling, etc.).
* May add more lines of code, complicating the overall structure.
* Appropriate refactoring techniques must be selected to address specific quality attributes, depending on the developers' experience and skills.

**Key Researchers**:

* Alshayeb (2009): Found that refactoring can improve some quality attributes while worsening others.
* Stroggylos and Spinellis (2007): Suggested that the choice of refactoring technique is crucial and should be guided by the developer's experience and skills.

**Detailed Explanation**:

* **Cohesion**: Refers to how closely related the functions within a single module are.
* **Coupling**: Refers to the degree of interdependence between software modules.
* **Balance**: Effective refactoring seeks to maximize cohesion and minimize coupling, but achieving this balance can be challenging.

**8. Refactoring Tools**

**Importance**:

* Refactoring tools in IDEs (e.g., Eclipse, Visual Studio, NetBeans) help automate refactoring tasks and ensure correctness.

**Benefits**:

* **Automation**: Tools automate repetitive refactoring tasks, reducing human error.
* **Guidance**: Provide suggestions and ensure best practices are followed.
* **Efficiency**: Increase productivity by simplifying complex refactoring tasks.

**Detailed Explanation**:

* **Eclipse**: Offers built-in refactoring tools like Rename, Move, and Extract Method.
* **Visual Studio**: Provides refactoring tools and integrates with tools like ReSharper for advanced refactoring.
* **NetBeans**: Includes a variety of refactoring tools for Java and other languages.