11 Refactoring - exam questions

You should know the answers to these questions:

Can you explain how refactoring differs from plain coding?

Refactoring is the process of changing a software system in a way that improves its internal structure without altering its external behavior. Plain coding focuses on adding new features or fixing bugs without necessarily improving the underlying code quality.

Can you tell the difference between Corrective, Adaptive and Perfective maintenance? And how about preventive maintenance?

- **Corrective Maintenance**: Fixing defects in the software.
- **Adaptive Maintenance**: Modifying software to work in new environments.
- **Perfective Maintenance**: Enhancing performance or adding new features.
- **Preventive Maintenance**: Improving code to prevent future issues.

Can you name the three phases of the iterative development life-cycle? Which of the three does refactoring support the best? Why do you say so?

1. **Prototyping**
2. **Expansion**
3. **Consolidation**

Refactoring supports the **consolidation phase** best because it focuses on improving the code structure after features have been implemented.

Can you give 4 symptoms for code that can be "cured" via refactoring?

1. Duplicated Code
2. Large Classes/Methods
3. Nested Conditionals
4. Abusive Inheritance.

Can you explain why add class/add method/add attribute are behaviour preserving?

These changes introduce new structures without altering existing behavior, ensuring external functionality remains unchanged while internal organization improves.

Can you give the pre-conditions for a "rename method" refactoring?

- Ensure no dynamic method calls depend on method names.
- Confirm tests cover all invocations of the method.
- Validate no external dependencies (e.g., configuration files) reference the method name.

Which 4 activities should be supported by tools when refactoring?

1. Source-to-source program transformation.
2. Regression testing.
3. Configuration and version management.
4. Reverse engineering capabilities.

Why can't we apply a "push up" to a method "x()" which accesses an attribute in the class the method is defined upon (see Refactoring Sequence on page 27–31)?

"Push up" cannot be applied because the method references attributes specific to its original class, and these attributes are not available in the superclass.

You should be able to complete the following tasks

Two classes A & B have a common parent class X. Class A defines a method a() and class B a method b() and there is a large portion of duplicated code between the two methods. Give a sequence of refactorings that moves the duplicated code in a separate method x() defined on the common superclass X.

1. Extract the common code into a new method `x()` in class X.
2. Replace the duplicated sections in `a()` and `b()` with calls to `x()`.

What would you do in the above situation if the duplicated code in the methods a() and b() are the same except for the name and type of a third object which they delegate responsibilities too?

1. Use the **Introduce Parameter** refactoring to make the differing object a parameter of the common method in class X.

Monitor the technical debt of you bachelor capstone project.

Use a version control system, static analysis tools, and manual reviews to track complexity and areas needing refactoring.

Can you answer the following questions?

Why would you use refactoring in combination with Design by Contract and Regression Testing?

- Design by Contract ensures preconditions, postconditions, and invariants are respected.
- Regression Testing verifies that refactoring does not break existing functionality.

Can you give an example of a sequence of refactorings that would improve a piece of code with deeply nested conditionals?

1. Apply **Extract Method** to isolate conditional branches.
2. Use **Replace Conditional with Polymorphism** if applicable.

How would you refactor a large method? And a large class?

- Large Method: Use **Extract Method** to break it into smaller, focused methods.
- Large Class: Apply **Extract Class** to split responsibilities into cohesive classes.

Consider an inheritance relationship between a superclass "Square" and a subclass "Rectangle". How would you refactor these classes to end up with a true "is-a" relationship? Can you generalise this procedure to any abusive inheritance relationship?

1. Apply **Replace Inheritance with Delegation** if a true "is-a" relationship does not exist.
2. Generalize: Identify common violations and apply either inheritance or composition as appropriate to ensure logical relationships.