EXAMINER'S COPY

EINDHOVEN UNIVERSITY OF TECHNOLOGY
Department of Mathematics & Computer Science

TU/e

Exam Programming Methods, 2IP15, Wednesday 17 April 2013, 09:00–12:00

THIS IS THE EXAMINER'S COPY WITH (POSSIBLY INCOMPLETE) ANSWER HINTS.

This *closed-book* exam consists of 16 questions, worth 3 points each, on 5 numbered pages. The final grade is computed as $(2 + s)/5$ rounded to one decimal, where s is the sum of the scores for the questions.

You must provide a to-the-point explanation or motivation for every answer.

1. What are the steps in the design technique *Divide & Conquer*?

   Hint Slide 24 of Lecture 1; three steps: split into subproblems, solve subproblems independently, compose subproblem solutions.

   For score ≥ 2, three distinct steps must have been described.

2. Give five advantages of a modular design over a monolithic design.

   Hint Slide 19 of Lecture 2; e.g., organize communication about design, facilitate parallel construction, help planning and progress tracking, improve verifiability, improve maintainability, improve reuse.

   For score 3, at least four distinct advantages must have been clearly described. Three, clearly described, distinct advantages, or unclear descriptions of four or more advantages lead to score 2. Score 1 requires at least two distinct advantages; less than two, gives score 0.

3. Give three guidelines for functional decomposition.

   Hint Slides 22–25 of Lecture 2.

   Score equals the number of distinct, clearly described guidelines.

4. Why is it important to reason about a method call in terms of the contract rather than the implementation of the method being called?

   Hint The implementation may realize more than was required by the contract. When the implementation is subsequently changed, the call may no longer work as intended if it relied on previous implementation details.

5. Are design patterns a form of code reuse?

   Hint No. A design pattern might provide sample code to help you understand the design, but it is the abstract design that is being reused not the concrete code. Rarely is the sample code provided with a design pattern reusable as is.

6. A program needs only one instance of class `GameTable`. Currently, the code looks like this (javadoc minimized):

```java
/** Game table with settable limit. */
public class GameTable {

    /** Limit for bets at this table (0 = no limit). */
    private int limit;

    /** Constructs a table without limit. */
    public GameTable() {
        this.limit = 0;
    }

    /** Sets the table limit. */
    public void setLimit(final int limit) {
        this.limit = limit;
    }

    // Other game table operations
    . . .
}
```

Apply the *Singleton* design pattern to ensure that a single object of this class is provided. It is to be used in a single-threaded application. Write out the Java code for the singleton version, and clearly indicate what code was added or modified.

Hint

```java
/** Game table with settable limit (singleton version). */
public class GameTableSingleton {

    /** ADDED: The single instance of this class. */
    private static GameTableSingleton instance;

    /** Limit for bets at this table (0 = no limit). */
    private int limit;

    /** MADE PRIVATE: Constructs a table without limit. */
    private GameTableSingleton() {
        this.limit = 0;
    }

    /** Sets the table limit. */
    public void setLimit(final int limit) {
        this.limit = limit;
    }

    /** ADDED: Gets the game table instance. */
    public static GameTableSingleton getInstance() {
        if (instance == null) {
            instance = new GameTableSingleton();
        }
        return instance;
    }

    // Other game table operations
    . . .
}
```

For score 3, the three added/modified elements must be correctly present (including **`static`**); some unnecessary additions/modifications can be tolerated, but within reason. Score 2 if the intention is clear, but there is a minor, easily repairable, mistake.

7. How is the *Iterator* design pattern a form of abstraction?

   Hint The *Iterator* design pattern abstracts from the nature of the collection (e.g. whether duplicates are allowed and whether order matters), the nature of the elements in the collection, and the mechanism for ensuring that each desired element is visited exactly once.

   Note that it is not the purpose of the iterator pattern to abstract from data representation; that is inherent in any abstract data type, with and without iterator(s).

8. You are working on a package with multiple classes, but would like to simplify the interface offered to clients of the package. What design pattern would be useful?

   Hint *Façade* design pattern

9. Are there advantages when implementing the *Adapter* design pattern through inheritance rather than composition?

   Hint Yes. When using composition, an extra object is created, which is typically an extra burden for the client code, and operations on the adapter object are delegated to the adapted object. This involves extra code and incurs some method call overhead. When using inheritance, there is only one object, no extra code, and method calls are direct. Furthermore, when using inheritance, protected members of the adapted class are available in the adaptation; this could lead to more efficient code.

   N.B. The implementation through composition is more flexible, however!

10. Why is it not a good idea to implement the *Decorator* design pattern through inheritance?

    Hint Because then you put a concrete decoration on one specific class, and cannot use various decorators together ("on top of each other"). The latter is possible when a decorator is implemented through composition. [Just stating that it is less flexible is not enough.]

11. Explain the relationship between the *Strategy* design pattern and the *Dependency Inversion Principle*. Draw a UML class diagram for the *Strategy* design pattern, and for its anti-pattern with direct dependency rather than inverted dependency.

    Hint See Slides 4–6 of Lecture 8, and the handout *From Callbacks to Design Patterns*.

    In the anti-pattern for the *Strategy* design pattern, client/context code of an algorithm, which is implemented in a concrete class, declares and uses a variable of that concrete class, and calls the algorithm directly. The client/context code depends on (cannot be compiled without) the concrete class.

    In the *Strategy* design pattern, a supertype (abstract class or interface) is introduced to provide only the signature of the algorithm method without any concrete implementation, and this supertype is used in the client/context code. At run-time, the client/context code can be handed an object of a concrete class that implements or extends the strategy supertype. The client/context code and the concrete algorithm classes both depend on the (more abstract) strategy interface/superclass. So, the dependency has been inverted.

12. How is the *Factory Method* design pattern a special case of the *Template Method* design pattern?

    Hint The creation of a new object is one step of a larger task. To make it possible for client code to vary the type of object created, the creation (`new` statement) is put in a separate overridable method, known as factory method, like a template method. Client code can subclass and override the factory method to create an object of a different type. The *Template Method* design pattern is applicable in more situations, where multiple subtasks of a task need to be overridable; these subtasks need not just concern object creation.

13. With the *Observer* design pattern, there are two ways of propagating data to observers: *push* and *pull*. What are the trade-offs?

    Hint The push approach requires subjects to be aware of what data the observers need. This increases coupling between subject and observers. Typically, too much data is sent, because not all observers have the same needs. The pull approach decreases this coupling but may be less efficient in other ways, because it will involve additional method calls on the subject to get data. The observers will need to find out what changes were made or assume anything could have changed. Note, in both cases the observers must have some knowledge of the subject they are observing (either to interpret the parameter passed in the update method or to query the subject for its updated state). Coupling can be reduced if the observers know the subject via an abstract class.

14. What design pattern has been used in the following piece of code? Give example code for a `ConcreteWaferProcessor` class, using appropriate `System.out.println(...)` for functionality.

```java
public abstract class AbstractWaferProcessor {

    /** Processes a given wafer. */
    public void processWafer(final Wafer wafer) {
        do {
            conditionWafer(wafer);
        } while (! measureWafer(wafer));
        exposeWafer(wafer);
    }

    /** Conditions a given wafer. */
    protected abstract void conditionWafer(Wafer wafer);

    /** Measures a given wafer, returning whether this was successful. */
    protected abstract boolean measureWafer(Wafer wafer);

    /** Exposes a given wafer. */
    protected abstract void exposeWafer(Wafer wafer);

}
```

Hint *Template Method* design pattern, where a concrete class overrides the template methods to vary the behavior without modifying the superclass; the client code still calls the `processWafer` from the superclass. Example:

```java
import java.util.Random;

/** A concrete named stochastic wafer processor. */
public class ConcreteWaferProcessor
        extends AbstractWaferProcessor {

    /** Oracle to decide measurement success. */
    private Random measureOracle = new Random();

    /** Name of this wafer processor. */
    private String name;

    /** Probability of successful measurement. */
    private double p;

    /** Constructs a concrete wafer processor with given name and
      * measurement success probability. */
    public ConcreteWaferProcessor(final String name, final double p) {
        this.name = name;
        this.p = p;
    }

    @Override
    protected void conditionWafer(final Wafer wafer) {
        System.out.println(name + " conditioning wafer " + wafer);
    }

    @Override
    protected boolean measureWafer(final Wafer wafer) {
        System.out.println(name + " measuring wafer " + wafer);
        return measureOracle.nextDouble() < p;
    }

    @Override
    protected void exposeWafer(final Wafer wafer) {
        System.out.println(name + " exposing wafer " + wafer);
    }

}
```

The instance variables and constructor are not required.

15. What roles are relevant in the *Composite* design pattern and how are they related? Draw a UML class diagram.

    Hint See Burris, extra chapter on Composite pattern; Slide 21 of Lecture 7. Roles: general component class, with two kinds of subclasses: non-container and container, where container subclasses hold references to zero or more component children.

16. When using the *Command* design pattern, the command objects respond to an `execute()` method and possibly also an `undo()` method. The preconditions of these methods are specific to the kind of command and its parameters. How are these preconditions guaranteed when using the *Command* design pattern to implement a multi-level undo-redo facility?

    Hint A multi-level undo-redo facility can be implemented through two stacks of commands: one undo stack of executed commands, and one redo stack of unexecuted (undone) commands. The commands on the undo stack appear in the order of execution, such that their preconditions are satisfied when applied in that order (otherwise, they would not have been executed in the first place; this is a responsibility of the controler). This also means that they can be undone in the reverse order. Similarly, commands on the redo stack appear in the order they were undone, and hence they can be (re-)executed safely in the reverse order. Briefly stated, the preconditions of the indicated methods are satsified because of the invariants for the undo and redo stacks.