EXAMINER'S COPY

EINDHOVEN UNIVERSITY OF TECHNOLOGY
Department of Mathematics & Computer Science

TU/e

Exam Programming Methods, 2IP15, Wednesday 26 June 2013, 14:00–17:00

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

This *closed-book* exam consists of 16 questions, worth 3 points each, on 4 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 abstraction mechanisms are available in the programming language Java to apply *Divide & Conquer*?

   Hint Slide 26 of Lecture 1: methods (procedural abstraction), classes (data abstraction), packages.

   For score ≥ 2, at least two mechanisms must have been described.

---

2. Why is it *not* a good idea to write monolithic programs? Give five reasons.

   Hint Slide 19 of Lecture 2; it hinders, e.g., communication about design, parallel construction, planning and progress tracking, verifiability, maintainability, reuse.

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

---

3. Why is it important to reason about a method implementation in terms of the contract rather than in terms of the actual calls of that method?

   Hint The actual calls may require less than was required by the contract. When subsequently new calls are introduced, these may not work as intended if the implementation relied on details of other calls.

---

4. Give three benefits of developing test cases *before* implementing a class and its methods.

   Hint See §3, page 4 of handout on TDD.

   Score equals the number of distinct, clearly described benefits.

---

5. What is an Abstract Data Type (ADT), and what does it take to implement an ADT?

   Hint Slides 16 and 22 of Lecture 4.

---

6. Given are the interfaces of a legacy and a Next-Generation wafer processor:

```
1 /** Interface of a legacy wafer processor. */
2 public interface IWaferProcessor {
3
4     /** Conditions a given wafer. */
5     void conditionWafer(Wafer wafer);
6
7     /** Measures a given wafer. */
8     void measureWafer(Wafer wafer);
9
```

```
10      /** Exposes a given wafer. */
11      void exposeWafer(Wafer wafer);
12
13  }
```

```
1   /** Interface of a Next-Generation wafer processor. */
2   public interface IWaferProcessorNG {
3
4       /** Loads and conditions a given wafer. */
5       void loadWafer(Wafer wafer);
6
7       /** Measures the loaded wafer. */
8       void measureWafer();
9
10      /** Exposes the loaded wafer, and returns it. */
11      Wafer exposeWafer();
12
13  }
```

Write a class `WaferProcessorAdapter` that adapts a legacy wafer processor (given as an object) to be usable through the Next-Generation interface.

Hint Because the class needs to adapt a given object, it must use composition rather than inheritance. Because the methods in the new interface do not all have the wafer as parameter, the adapter must store it locally upon loading the wafer.

```
1   /** Adapts legacy wafer processor to Next Generation interface. */
2   public class WaferProcessorAdapter implements IWaferProcessorNG {
3
4       /** Wafer processor being adapted. */
5       WaferProcessor adaptee;
6
7       /** Loaded wafer. */
8       Wafer wafer;
9
10      /** Constructs an adapted wafer processor for adaptee. */
11      WaferProcessorAdapter(WaferProcessor adaptee) {
12          this.adaptee = adaptee;
13      }
14
15      @Override
16      public void loadWafer(Wafer wafer) {
17          this.wafer = wafer;
18          adaptee.conditionWafer(wafer);
19      }
20
21      @Override
22      public void measureWafer() {
23          adaptee.measureWafer(wafer);
24      }
25
26      @Override
27      public Wafer exposeWafer() {
28          adaptee.exposeWafer(wafer);
29          Wafer result = wafer;
```

```
30            wafer = null; // not required
31            return result;
32        }
33
34    }
```

7. 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.]

8. In what ways is a singleton class better than using just a global variable?

   Hint See Burris, Ch. 2

9. Explain the *Iterator* design pattern.

   Hint See Burris, Ch. 3

10. When is it appropriate to apply the *State* design pattern?

    Hint See Burris, Ch. 6, page 76.

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. What is the intent of the *Factory Method* design pattern?

    Hint See Burris, Ch. 8.

13. Why is it useful that the update method in the Observer interface includes a reference to the subject (the object being observed)? Does the observer not know what object it is observing?

```
1     interface Observer {
2         void update(Subject subject, Data data);
3     }
```

    Hint The observer may be observing more than one subject using the same Observer interface. The subject parameter identifies the object which has been updated, which may be needed to pull more data from it.

EXAMINER'S COPY

14. 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.

---

15. 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.

---

16. Give three benefits of applying design patterns.

    Hint See Burris, pages 19–22.

    Score equals the number of distinct, clearly described benefits.
