Software Engineering Design & Construction
In The Past

Time for Solving the Exam	90 Minutes
Permitted Aids........................	**Everything except electronic devices**
Examiner.................................	Dr.-Ing. Michael Eichberg

Remarks:

- Only use black or blue, document-proof pens. Answers given using a non-conformant pen will be ignored and will not be graded.
- Do not remove the stapling.
- Put your name and registration number on each additional paper that you have requested.
- Write your name in block letters.
- Write legibly.
- Completely fill out this page and do not forget to sign the exam.
- Incorrect answers may lead to a subtraction of points.
- **The post-exam review will be announced in the forum.**

1. General Knowledge Questions

1.1. Design

Why can a benign change of some (domain) requirement result in significant effort on part of the developers.

From a software design point-of-view: Why do we develop new language features?

1.2. Clean Code

The principle of least astonishment is a guideline for writing clean code. Explain the idea of this principle in one sentence.

What does the Boy-Scout Rule mean in the context of writing clean code?

Explain the Don’t-Repeat-Yourself (DRY) principle in one sentence.

1.3. Testing

Why do we consider “Testing” to be part of the design?

Name the phases of a Four-Phase-Test and explain each one with a single sentence.

Explain why fully automated test suites with high functional coverage are essential for software development.

In the context of testing, name an indirect input that can cause issues with fully automated test-suits.

Why are indirect inputs a problem for testing?

2. Design Principles

It is possible to avoid polluted interfaces by complying with the Interface-Segregation Principle. Describe why this is an advantage.

Helper classes (e.g. `java.lang.Math`) that offer a bunch of static helper methods are common in Java. Name one design principle that is often violated by such classes. Briefly justify your answer.

The Open-Closed Principle is closely related to Agile Development: Being closed against changes and open for extension allows you to react on change requests. Does this mean that you need to anticipate all changes you can possible think of? Justify your answer.

Does the Strategy Pattern support the Single-Responsibility Principle? Justify your answer.

Name two Design Principles that also apply at the package level. Shortly explain in which way they are related to package-level design decisions.

3. Design Patterns

Name three Design Patterns from the lecture that apply the Dependency-Inversion Principle (DIP). Justify each choice with a single sentence.

Name two language features of object-oriented programming languages, such as Java or Scala, that support the Open-Closed Principle (OCP)? Give a short justification for each feature.

3.1. Differentiate Patterns

Explain the difference in what the Object-Adapter Pattern and the Decorator Pattern achieve.

Explain the difference between the Decorator Pattern and the Proxy Pattern from the client's point of view.

3.2. Visitor Pattern

Shortly explain what Dynamic Dispatch is.

Shortly explain what Double Dispatch is.

Double Dispatch is not natively supported by Java. Explain how the Visitor Pattern simulates it and why this works.

4. Designing an API

For this task you are going to design different aspects of an API for a hierarchical file system. Develop a design for a pure Java application.

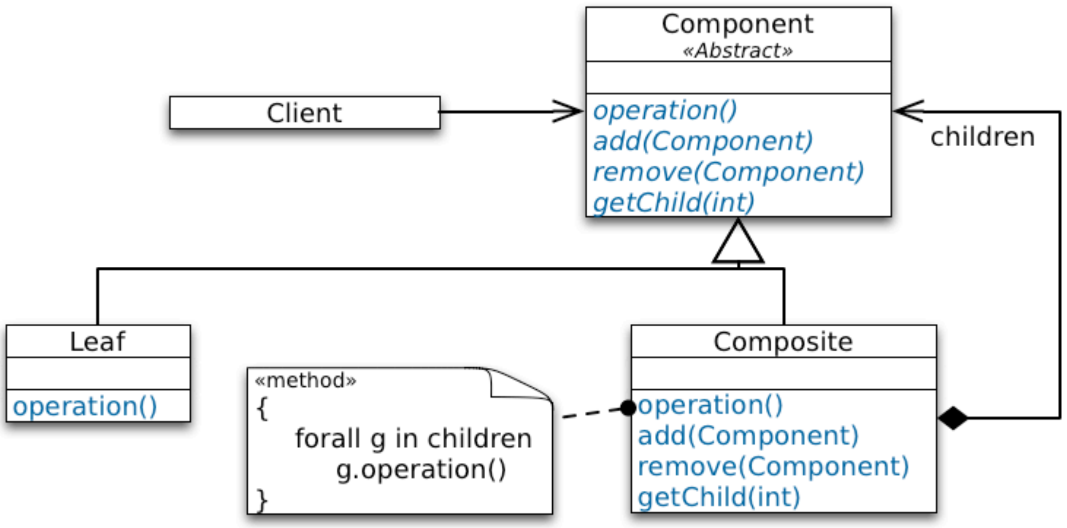
From the point of view of a developer using your API, a file system should handle file structures of arbitrary size and complexity. It shouldn't put arbitrary limits on how wide or deep the file structure can get. Also the API should be easy to deal with and to extend.

Hint: When you draw UML diagrams you don't have to repeat all field and method declarations. It suffices if you add those declarations that are relevant for the specific.

4.1. Composite

The basic elements stored in a file system are files and directories. These elements are the nodes of the file system hierarchy. Files are nodes that store some data as a byte stream. Directories are nodes that can contain files and other directories. For many operations, like changing access rights, renaming, or moving, we want to treat both kinds of nodes in the same way.

In books on Design Patterns you find the "Composite" pattern, which is intended to *"compose objects into tree structures to represent part-whole hierarchies. [It] lets clients treat individual objects and compositions of objects uniformly."* The pattern's abstract structure is:



Propose a design for representing hierarchical file systems that applies the Composite design pattern. Draw a UML class diagram that depicts the relations between the different classes. You don't need to define any concrete operations on file-system nodes.

Assume that a field "name" is defined on all nodes. Write an algorithm that recursively searches for all nodes with a given name. The algorithm should not have to distinguish files and directories. You can use Java-style pseudo-code for this task.

```
public (Set of Nodes) find(queryName, node) {

   …



}
```

4.2. Symbolic Links

A symbolic link is a node in the file system that refers to another node in the file system. From the user's point of view a symbolic link acts like a synchronized copy of the referenced node, with the exception that, if you delete a link, the referenced node remains untouched.

Such links can be integrated into the existing design by use of a pattern.

Name the pattern in question.

Draw a UML class diagram that displays the design related to the link feature and how it is related to the overall design.

Look at your changed design from the viewpoint of the Composite Pattern. What are the children of link nodes?

Does the symbolic-link extension break your search algorithm when linked directories should also be searched? Briefly justify your answer.

4.3. File Operations

From now on, consider the design of the class hierarchy for the representation of the file system as stable (i.e., no classes will be added or removed from the hierarchy).

By which means can we achieve extensibility for new commands, like renaming, deleting, or changing rights, without changing existing code?

Draw a UML class diagram that depicts the extended design. Do not forget to include methods that are necessary to fully comprehend the design.

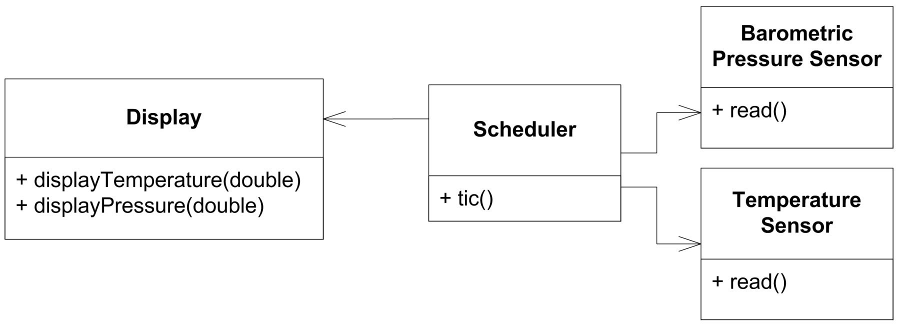
A common operation on a file system is to list all files in a given directory as well as all files in all subdirectories. Implement this operation using your proposed design. Do not follow links to avoid having to deal with cyclic structures. You can use `println` as a shorthand for `System.out.println`. The suffix of directories should be '/' and the suffix of links should be '-> «name of the referenced node»'.

5. Find and Resolve Design Problems

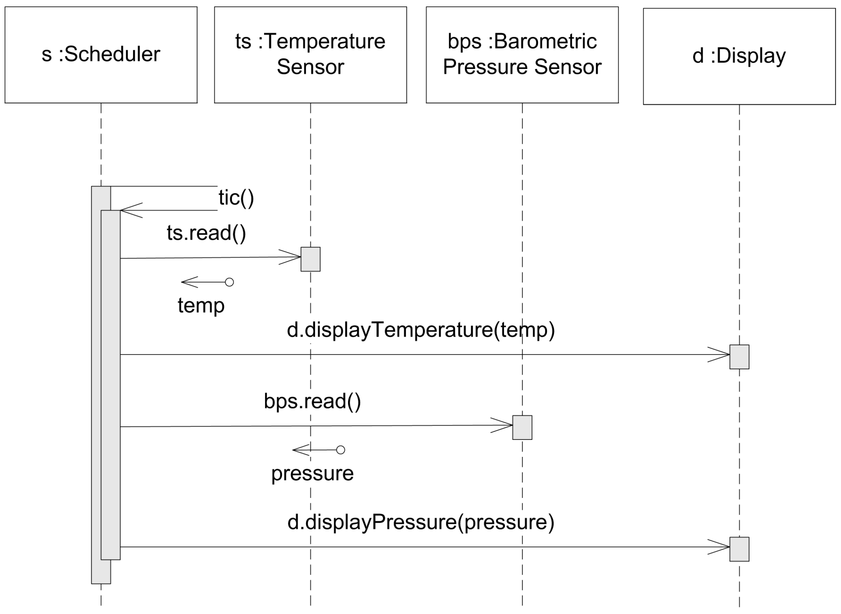
Your company wants to bring a meteorological station onto the market. The basic model consists of two sensors that measure air pressure and temperature and show the respective values on a display.

There are some new sensor and display technologies in the pipeline that were not yet disclosed, but you should be able to easily integrate them later on.

The company's architect sent you his design draft:


Display
+ displayTemperature(double)
+ displayPressure(double)
Scheduler
+ tic()
Barometric Pressure Sensor
+ read()
Temperature Sensor
+ read()

Furthermore, he sent you a sequence diagram that shows the control flow between the components. He tells you that the `Scheduler` reads the sensors every second and delegates the respective values to the display. Hence, the scheduler manages the timing, the querying of the sensors, and the update of the display. The class `Display` is responsible for representing the values on the monitor.


s :Scheduler
ts :Temperature Sensor
bps :Barometric Pressure Sensor
d :Display
tic()
ts.read()
temp
d.displayTemperature(temp)
bps.read()
pressure
d.displayPressure(pressure)

The following code snippet demonstrates the initialization of the station:

```
public void initialize() {
  TemperatureSensor ts = new TemperatureSensor();
  BarometricPressureSensor bps = new BarometricPressureSensor();
  Display d = new Display();
  Scheduler s = new Scheduler(ts, bps, d);
}
```

Various design principles are violated by the proposed design. Name three violated principles and explain each violation in a single sentence.

Describe how you would remove each of the identified violations. It is not necessary to create a refactored design model for this task, just describe the approach. Make sure that the connection between violation and applied refactoring is clear.

Since a lot of work has already been spent in the development of the display and sensor implementations (part of which was outsourced to another company), these classes cannot be changed anymore. Therefore, it was proposed to decouple the `Scheduler` from the rest of the system, using an `SchedulerSensorInterface` (see diagram below).

Which Design Pattern is used in this new design?

Extend the given UML diagram. Add the necessary classes and connections between the classes and name the roles the classes have w.r.t. the implemented pattern. Mind your own advice from before.

Change the initialization code snippet below to conform to the new system design. Also sketch an implementation for the adapter.

```
public void initialize() {
  TemperatureSensor ts = new TemperatureSensor();
  Display d = new Display();
  Scheduler s = new Scheduler();



…



}
```

```
public class TemperatureSensorAdapter … {



…



}
```

Is the new design closed against the changes that were anticipated by the Cloud Company?

6. Covariance and Contravariance

Consider the following type hierarchy from the Scala Collections Library:

```
ArrayBuffer[T] <: Seq[+T] <: Iterable[+T] <: AnyRef <: Any
```

given the function definition

```
def identity(l: Seq[AnyRef]): Iterable[AnyRef] = l
```

which of the following lines are not type-safe, i.e., which do not compile:

```
val f1: (Seq[Any]) => AnyRef = identity
val f2: (Iterable[AnyRef]) => Iterable[AnyRef] = identity
val f3: (ArrayBuffer[AnyRef]) => AnyRef = identity
val f4: (ArrayBuffer[Any]) => Iterable[AnyRef] = identity
```

For each line that does not compile briefly explain the reason.

7. Aspect-oriented Programming (AOP)

Explain the term Crosscutting Concern.

Describe a typical example of a crosscutting concern in applications that don't use aspect-oriented programming languages. Explain why scattering occurs in that case and how AOP languages solve the problem.