2021-12-16-Software Systems - Design

Course: B-CS-MOD02-1B-202001024 B-CS Software Systems Core 202001024

Duration:	3 hours
Generated on:	Jan 3, 2022

Contents:

Pages:

2021-12-16-Software Systems - Design

Course: B-CS-MOD02-1B-202001024 B-CS Software Systems Core 202001024

Welcome to the Software Systems Design **digital exam**:

- This is an open book exam. You are allowed to use the slides. The slides are accessible on your Chromebooks under the whitelisted URL (http://grammarware.net/slides/2021/ss/) **Note:** A tab with the slides is already open by default.
- **Smartphones or personal notebooks are not allowed. Put those in your bag now (with the sound switched off)**
- For **technical questions** concerning the chromebooks, Remindo, log-in issues etc.: raise your hand.
- For **content questions**: use the BBB chat.
- Do not forget to **save** your answers once you are done.
- There are **12 questions** and **60 points**.

Good luck and enjoy the test!

1 "All models are wrong" -- This sentence was used repeatedly throughout the course.
3 pt. Briefly explain the meaning of the statement in your own words.

2 Recall UML State Machine Diagrams.

Can you use them as (a.) descriptive, (b.) predictive, and (c.) prescriptive models?

For each of them give an answer and shortly motivate it.

2 pt. **a.** Descriptive:

2 pt. **b.** Predictive:

2 pt. **c.** Prescriptive:

3 Within the framework of Test Driven Development (TDD), what is the consequence of specifying too
2 pt. few test cases? Sketch at least two issues.

4 Given below are pairs of classes (eg. "House" and "Room").

For each pair:

1. define a sensible association between the classes (eg. "lives in" or "composition"),
2. specify the multiplicity/cardinality of the association if appropriate (eg. "1 - 1" or "1..* - 0.. 5").

2 pt. **a.** "Laptop"
"Device"

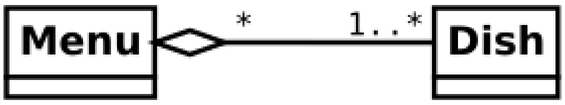
2 pt. **b.** "House"
"Landlord"

2 pt. **c.** "Java Program"
"Class"

2 pt. **d.** "Key"
"Keyboard"

2 pt. **e.** "Woman"
"Human"

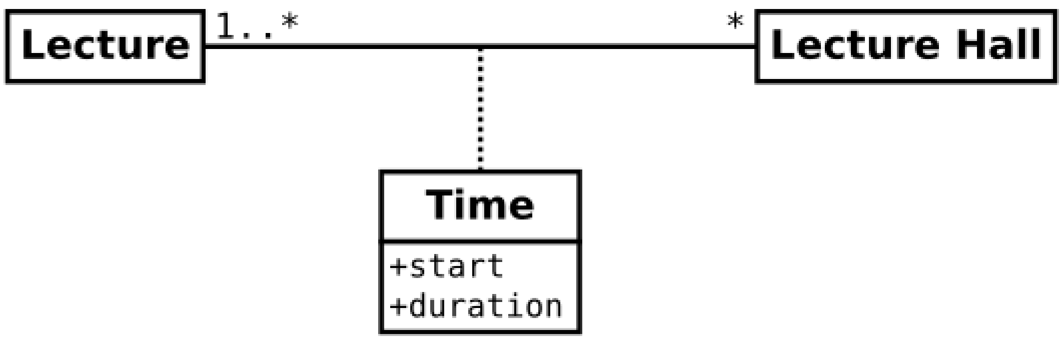
5 Consider the following class diagram:
4 pt.



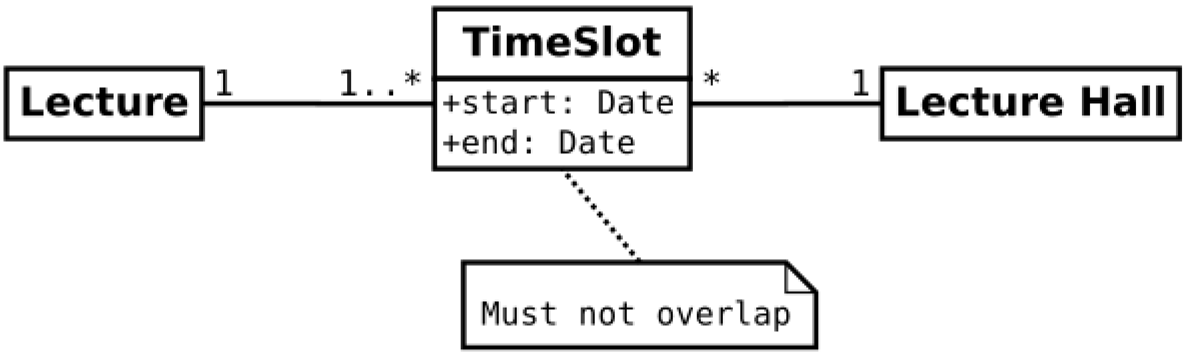
Specify two meaningful attributes for each class. Include visibility and type.

6 Consider the following two class diagrams, that model the schedule of lectures:
6 pt.

Class Diagram 1:



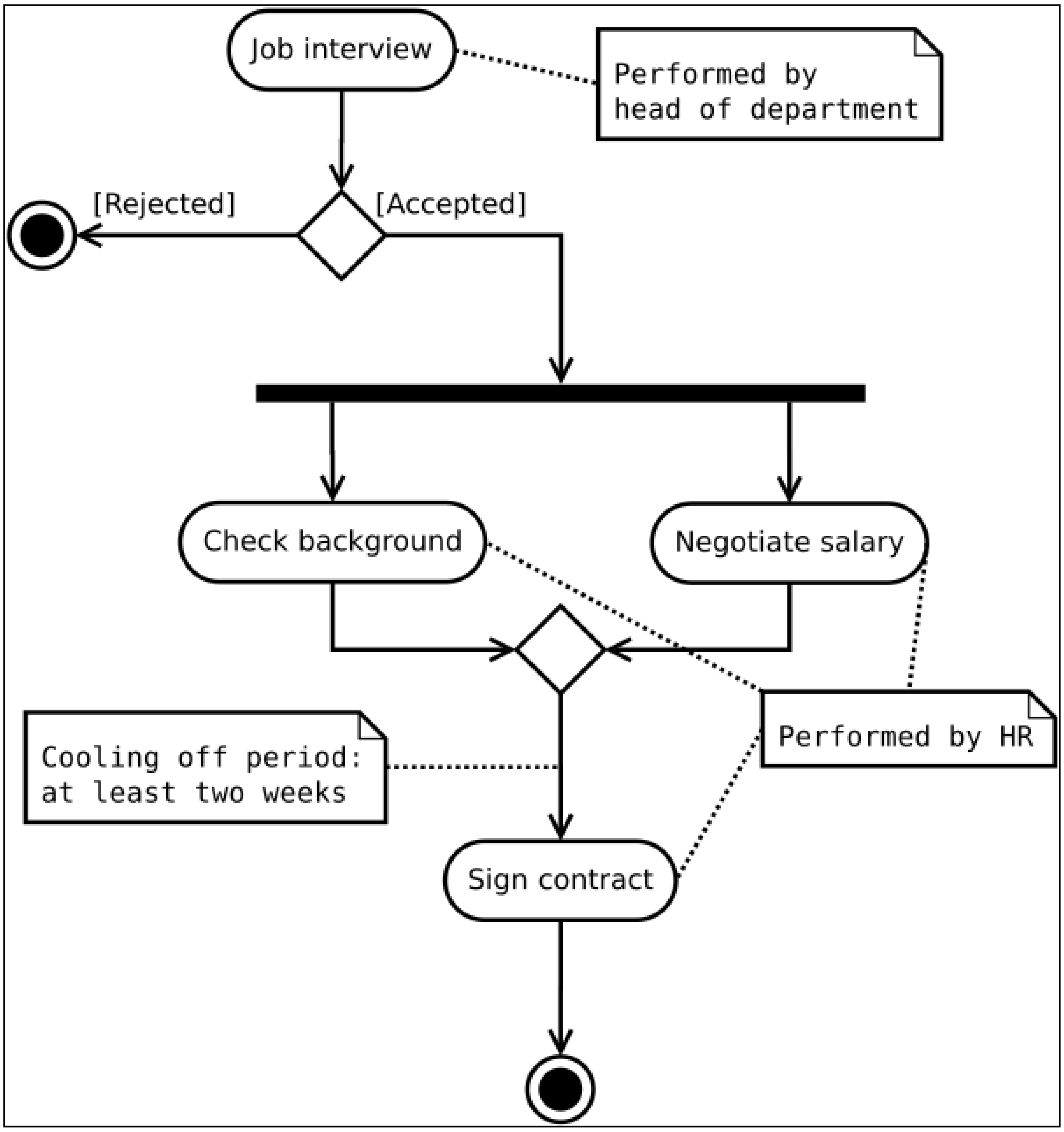
Class Diagram 2:



Compare the two diagrams. Point out at least three advantages/disadvantages of the models.

7
4 pt.

Given is the following Activity Diagram:



Name two properties of the diagram you would improve, and shortly explain how you would improve them.

8
2 pt.

A developer in your project group has the habit of committing once per day, no matter the state of their local copy.

As commit message they always use "today's work :)".

Why is this a bad idea?

9 Are the following good Software Metrics?

Explain your answer.

2 pt. **a.** Number of methods in a class.

2 pt. **b.** Average number of methods per class in the project.

2 pt. **c.** Number of (other) classes that a class depends on.

2 pt. **d.** Number of commits per developer.

2 pt. **e.** Number of methods in a project whose names contains the letter 'b'.

10 3 pt. You are a software developer at a company, developing and maintaining an application built in Java. Your team lead asks you to migrate from Swing to JavaFx, so that instead of JButton you'll use Button, instead of JLabel you'll use Label, etc.

Your colleague proposes to start with a global search-and-replace, automatically removing all occurrences of the letter J from the codebase.

Give at least three examples of what could possibly go wrong.

11 The following guidelines were given to students that want to write good code:

1. Write short methods
2. Give meaningful names
3. Write simple methods
4. Write code once
5. Keep signatures small
6. Separate concerns in classes
7. Keep components balanced
8. Keep the code clean
9. Automate everything
10. Keep improving the code

For each of them find a legitimate excuse to break the rule locally, or explain why the guideline is actually a universally unbreakable rule.

For example:

- A legitimate excuse to break rule 9: "This program will definitely only be run once with these parameters, so there is no need to add them to a makefile."
- A reason why rule 10 is a universally unbreakable law: "Software that is not being improved becomes legacy".

1 pt. **a.** Write short methods.

1 pt. **b.** Give meaningful names.

1 pt. **c.** Write simple methods.

1 pt. **d.** Write code once.

1 pt. **e.** Keep signatures small.

1 pt. **f.** Separate concerns in classes.

1 pt. **g.** Keep components balanced.

1 pt. **h.** Keep the code clean.

12 2 pt. As it was discussed in the lecture, the five essential elements of modular software design are

1. purpose,
2. connection,
3. interface,
4. encapsulation, and
5. implementation.

Pick one, briefly describe what it means, and argue why it is important.

Thank you, your answers were saved. You will be informed about your grade after the exam correction.

Take care!

Correction model

1.
3 pt.

Correction criterion	**Points**
A model is an abstraction: It focuses on some details while neglecting others. It will be wrong in any situation in that the neglected details are of importance. If the model adequately captured *all* important details, it would not be a feasible model.	3 points
Total points:	*3 points*

2.
6 pt.

a.

Correction criterion	**Points**
Answer: We accept all reasonings that show that the student has understood the difference between D/P/P modelling and knows what an SMD is Descriptive: Yes. - For example SMD can be used to describe the navigation inside a website. - Automata learning uses SMDs to learn the behaviour of a given system. etc. In particular, "no"-answers can be correct too, if the student shows expertise in SMD and D/P/P.	2 points
Total points:	*2 points*

b.

Correction criterion	**Points**
Answer: We accept all reasonings that show that the student has understood the difference between D/P/P modelling and knows what an SMD is Predictive: Yes, after learning the behaviour of a given system it can be used to make predictions for other inputs. In particular, "no"-answers can be correct too, if the student shows expertise in SMD and D/P/P.	2 points
Total points:	*2 points*

c.

Correction criterion	**Points**
Answer: We accept all reasonings that show that the student has understood the difference between D/P/P modelling and knows what an SMD is Prescriptive: Yes, for instance when used in a software design lifecycle an SMD might prescribe how to implement the final product. In particular, "no"-answers can be correct too, if the student shows expertise in SMD and D/P/P.	2 points
Total points:	*2 points*

3.
2 pt.

Correction criterion	**Points**
We accept, any reasoning that shows the student understands what TDD is. For example, specifying too few test cases: - ... results in confused programmers, that don't know how to resolve ambiguities - ... no more code than needed to pass (TDD law). This could result in incomplete code. - ... results in huge test cases to test a lot of functionality - ... would result in errors going undetected (Note: This is not directly linked to TDD itself, but requires some context) In particular, platitudes like "this is bad practice and should be avoided", or "this results in bugs" are given 0 pts	2 points
Total points:	*2 points*

4.
10 pt.

a.

Correction criterion	**Points**
- Generalisation: - Laptop is a Device - multiplicity does not apply	2 points
Total points:	*2 points*

b.

Correction criterion	**Points**
- Association or Aggregation - House has a landlord - 1.. * to 1	2 points
Total points:	*2 points*

c.

Correction criterion	**Points**
- Aggregation or Composition - Java program has a class - 0.. * to 1.. *	2 points
Total points:	*2 points*

d.

Correction criterion	**Points**
- Composition: - Key part of keyboard - 1.. * to 1	2 points
Total points:	*2 points*

e.

Correction criterion	**Points**
- Generalisation: - Woman is a human - Multiplicity does not apply	2 points
Total points:	*2 points*

5.
4 pt.

Correction criterion	**Points**
Many answers are possible, for instance: Menu: - restaurant name: String - dish of the day: Dish - name of the owner: String - etc. Dish: - price: Real - name: String - ingredients: String[1..*] - etc.	4 points
Total points:	*4 points*

6.
6 pt.

Correction criterion	**Points**
Anything sensible will be accepted. For example: (Note: For the answers below, you can argue for the opposite. It is the argument that counts.) - A good comment can significantly clarify the model, unfortunately it's missing in CD1. - A comment may be interpreted in a way that is not intended. - Association classes allow for more concise modeling. - Mismatch between multiplicities in CD1 and CD2: In particular, CD1 does not allow for lecture halls without lectures, - Mismatch between multiplicities in CD1 and CD2: In particular, CD2 does not allow for virtual lectures (without lecture halls) - Types can clarify implementation oriented modelling, but may obfuscate a more abstract one; in particular if the attribute names clearly indicate the types - Modeling "start" and "duration" does not allow for corner cases like "end time" before "start time". - CD2 Timeslot -> Time - CD2 requires to be able to parse the comment in human language.	6 points
Total points:	*6 points*

7.
4 pt.

Correction criterion	**Points**
For example, possible answers are: - Add a start node - Introduce swim lanes for different actors - proper syntax use of merge & fork - add hourglass for cooling off period Note that the question requires to comment on the activity diagram, not the underlying process in the company.	4 points
Total points:	*4 points*

8.
2 pt.

Correction criterion	**Points**
There are many issues with this commit strategy. For example, commits should: - [...] focus on single changes - [...] not be done on partial changes that make the project not compile - [...] be done frequently, so others can continue their work - [...] never commit broken code - Commit messages should be meaningful, such that others can see what the commit was about	2 points
Total points:	*2 points*

9.
10 pt.

a.

Correction criterion	**Points**
Yes. Might hint at bloated classes.	2 points
Total points:	*2 points*

b.

Correction criterion	**Points**
Yes. Similar to first, but gives overall indication.	2 points
Total points:	*2 points*

c.

Correction criterion	**Points**
Yes. Might hint at the quality of modular design (coupling).	2 points
Total points:	*2 points*

d.

Correction criterion	**Points**
Yes. Provided developers use a sensible commit strategy. Gives a good overview of project structure and collaborators/activity and responsibility of developers.	2 points
Total points:	*2 points*

e.

Correction criterion	**Points**
No. Seems completely arbitrary	2 points
Total points:	*2 points*

10.
3 pt.

Correction criterion	**Points**
Many answers are possible. Below we list a few: - Some concepts are not the same by just removing the letter J, eg. JFrame should be Stage and JPanel should be Scene, etc - Methods will be different , eg. .show() instead of .setVisible(true), .putConstraint spread out across calls, etc. - replacing visitors with event handlers is a redesign that cannot be solved on a textual level - We possibly need to import different libraries - the letter J can occur in text messages and other unrelated code fragments Answers that are not accepted are similar to those below: - This should not be done - Migration is unnecessary - etc.	3 points
Total points:	*3 points*

11.
8 pt.

a.

Correction criterion	**Points**
It can be argued for or against -- It's the argument that counts. For instance: - Can be broken locally if a method requires a lot of step-by-step mathematical calculations	1 point
Total points:	*1 point*

b.

Correction criterion	**Points**
It can be argued for or against -- It's the argument that counts. For instance: - Can be broken locally if the only way to give a meaningful name is by using very long names that obfuscate from the functionality.	1 point
Total points:	*1 point*

c.

Correction criterion	**Points**
It can be argued for or against -- It's the argument that counts. For instance: - The understanding of "simplicity" may vary with whoever reads it. This can be broken locally.	1 point
Total points:	*1 point*

d.

Correction criterion	**Points**
It can be argued for or against -- It's the argument that counts. For instance: - This rule can temporarily be broken, if we re-implement a method from ground up to improve it.	1 point
Total points:	*1 point*

e.

Correction criterion	**Points**
It can be argued for or against -- It's the argument that counts. For instance: - This is an unbreakable rule; if the number of arguments becomes too large, it is time to split the functionality of the method.	1 point
Total points:	*1 point*

f.

Correction criterion	**Points**
It can be argued for or against -- It's the argument that counts. For instance: - Unbreakable rule: It is always possible to separate concerns into classes. If this is not possible, then the class is sufficiently separated.	1 point
Total points:	*1 point*

g.

Correction criterion	**Points**
It can be argued for or against -- It's the argument that counts. For instance: - Can be broken locally as occasionally the structure of a program requires one component to be vastly smaller than others.	1 point
Total points:	*1 point*

h.

Correction criterion	**Points**
It can be argued for or against -- It's the argument that counts. For instance: - Unbreakable law: Clean code means it's clear to read and clear to change. Conversely, having messy code means it becomes unmaintainable.	1 point
Total points:	*1 point*

12.
2 pt.

Correction criterion	**Points**
Every element can be argued for: The following answers are taken from: https://www.genui.com/resources/5-essential-elements-of-modular-software-design Purpose: A module is an abstraction with purpose. Its purpose should be clear. It should have a single, exclusive responsibility. A module's responsibility should be narrow and focused, and no two modules' purpose should overlap. Connection: The connections between modules adds their own complexity to the overall system. A well designed modular system minimizes the dependencies between modules. Minimize each module's external dependencies and periodically review the overall modular-design to look for complexity-decreasing opportunities. Interface: A module's interface should be easy to use, easy to understand and easy to ensure correctness. It should offer all this without needing to understand any of its implementation details. To achieve this, a module's API should be well-defined and documented. The API should be complete and minimal. It should have exactly what is needed and nothing more. Last, it should be hard to misuse. The easiest way to use a module should also be the correct way. Encapsulation: A module's implementation is private. Modules should expose as little as possible. They should not expose their functional structure, data-structure nor their own dependencies. Any implementation detail of a module should be changeable without affecting a single client. This is perhaps the most important element of modular design. The other four elements could all be expressed in terms of maximizing the isolation as much as possible of the internal implementation from the outside world. As an abstraction, each module should as watertight as possible. Leaks become accidental, hidden parts of the public API. Without strong encapsulation, you end up with implicit dependencies which can be disastrous to scaling projects. Implementation: In order to ensure the module is easy to use, it must work well. A module with the best-designed purpose, interface and encapsulation will still fail without good implementation. A module's implementation should be correct, performant, tested and minimal. The worse the implementation, the leakier the abstraction.	2 points
Total points:	*2 points*

Caesura

Points scored	**Grade**
60	10
59	9.8
58	9.7
57	9.5
56	9.3
55	9.2
54	9.0
53	8.8
52	8.7
51	8.5
50	8.3
49	8.2
48	8.0
47	7.8
46	7.7
45	7.5
44	7.3
43	7.2
42	7.0
41	6.8
40	6.7
39	6.5
38	6.3
37	6.2
36	6.0
35	5.8
34	5.7
33	5.5
32	5.4
31	5.2
30	5.1
29	5.0
28	4.8
27	4.7
26	4.5
25	4.4
24	4.3
23	4.1
22	4.0
21	3.9
20	3.7
19	3.6
18	3.5
17	3.3
16	3.2
15	3.0
14	2.9
13	2.8
12	2.6
11	2.5
10	2.4
9	2.2
8	2.1
7	2.0
6	1.8
5	1.7
4	1.5
3	1.4
2	1.3
1	1.1
0	1.0

Question identifiers

These identifiers can be used to track the exact origin of the question. Use these identifiers together with the identifier of this document when sending in comments about the questions, so that your comment can be connected precisely with the question you are referring to.

Document identifier: 4815-10053

Question number	Question identifier	Version identifier
1	43224	49c31bb2-81cc-0bfb-ea83-f8efbae922f2
2	80207	e95e4713-a59d-30e6-74e1-89ab3a653e42
3	80212	9f621441-ab83-b7f7-6248-1728672f8719
4	80487	48726fec-4ae6-54ac-eff3-aa98bdd77324
5	80492	746c75e8-19a4-a00f-0cfb-b08295d70790
6	80217	f88aefc0-e093-2949-0cc8-9e1632ddc6dd
7	80222	29e96949-6883-e5cf-200a-c1307aa5a091
8	80227	92b1c781-17ee-6dd6-f6f3-29d2e01da03e
9	80237	ec5ac82f-1f44-5dc0-cc6c-1e4ca1595cb2
10	80537	b2f1e291-5d8d-8d04-85e3-ddb1ae5b5d14
11	43272	6cfb570b-4bac-ed1a-812a-04d5ec0a4ed5
12	80242	2cfc8aba-a639-f1e4-6429-b6a521f52aa9