How to pass the SWEA exam

Henrik Bærbak Christensen

Status: Release 11-2025.

November 25, 2025

Chapter 1

SWEA Exam

In this document, I will provide insight into the core aspects that censor and examiner use to base their evaluation and grading of the SWEA exam, and give hints for how to prepare yourself best for the exam.

1.1 Examination Areas

The questions are grouped witin the following areas:

- *Test-driven development.* Emphasis on applying the rhythm and using/understanding the values and TDD principles.
- *Systematic black-box testing.* Emphasis on applying and understanding equivalence partitioning techniques and boundary value analysis.
- *Variability management.* Emphasis on applying the four different techniques for handling variability and analysing their benefits and liabilities.
- *Test stubs and unit/integration testing.* Emphasis on applying test stubs and understanding the testing levels of unit/integration/system testing.
- *Design patterns.* Emphasis on finding the proper design pattern for a problem at hand and applying it.
- *Compositional design.* Emphasis on applying compositional design principles and relating it to associated concepts like ISP, role- and private interfaces, and multi-dimensional variance.
- *Frameworks.* Emphasis on designing frameworks and understanding framework theory.
- *Clean Code and Refactoring.* Emphasis on properties of clean code and using refactoring to improve analyzability.
- *Distribution and Broker.* Emphasis on designing distributed client-server architectures using the Broker pattern.

1.2 General Advice

Remember this is an exam where you are required *to be able to apply techniques.* Metaphorically, I am not much interested in hearing about how to do push-ups ("First I bend my arm and then I extend it again"), I am interesting in seeing you actually doing push-ups. So, be sure to train applying the techniques in code—code the Strategy pattern, refactor it, and code it again; test-drive the TDD question code examples, apply the equivalence class testing process for the demo questions or some examples in the book, etc.

Key Point: Train your ability to apply the techniques

At the exam you must demonstrate your ability to apply techniques on the concreete exam question. A purely theoretical overview that does not attempt to solve the given exercise will not pass the exam.

All questions have a concrete part (the given exercise) and a request to broaden the discussion to a more theoretical level. You may choose to start your presentation at "both ends". For instance, if a particular problem requires the use of the State pattern you may either describe the concrete solution to the exercise first and then relate to the more general description of State pattern; or you may describe State in general terms first and then show how it has been applied to solve the concrete problem. Remember, however, that 12 minutes is a short time: for "top" presentations I expect both aspects are treated.

Key Point: Practice first or Theory first

You may solve the exercise first and relate to theory; or you may provide a short theory overview first before solving the exercise. Be sure to have enough time to solve the exericse, though.

As you know the examination areas in advance and know there should be a more theoretical discussion this is a part you can prepare to some detail in advance. I advice that you use this opportunity.

Key Point: Prepare your theory presentation in advance

As you know the examination areas in advance, you have the chance to select and rehearse the central concepts and terminology in advance: take this chance!

Please note that emphasis is put on the operational level when considering passing/not passing the exam! By this I mean that you must demonstrate the ability to write correct Java (or another OO language) code that implements the basic techniques presented in the course. I have quite a few students that sadly fails the exam, because they are not able to write simple things like instantiating an object in Java.

Key Point: Train your coding abilities

SWEA is a course on advanced ***programming techniques.*** *If you cannot demonstrate sufficient basic programming skills (ala IntProg skills), then of course you cannot pass an advanced programming course!*

As you will be presenting code using Zoom's Shared Screen of your laptop it is essential that you find good tools that you are comfortable with, and do not get in the way. Word is terrible to present code (you will spend most of your time adding spaces to do indentation); and drawing a UML diagram using the mouse in a paint program is a disaster. IntelliJ may seem the right choice for coding, but I fear it will complain too much—some of the code will be pseudo code. So pick a good programming editor that is not too fuzzy, like GEdit, NodePad++, Sublime Text, etc.

Key Point: Be comfortable with your selected Tools

Find good code- and diagram editors to allow you to focus on presenting code, diagrams, tables, etc., without struggling with the programs themselves.

Finally, remember to **test the technical setup** before the exam! Mac users often forget to enable screen sharing (forcing reboot and panic), and several Ubuntu versions does not support screen sharing at all (forcing panic and using the whiteboard).

Key Point: Test Screen sharing over Zoom

Clean technical setup that works!

Chapter 2

Advice for Each Examination Area

2.1 Regarding Test-driven development

2.1.1 Regarding the theory

The central TDD theory consists of a) the values b) the rhythm and c) the TDD principles. And these again rely on the testing terminology of FRS Chapter 2.

It makes a good impression to be able to explain the rhythm and some of the central principles and especially if you can relate them to the values of TDD.

2.1.2 Regarding solving the exercise

Be sure to read the exercise so you solve the right exercise and not what you think is the exercise.

Do not make an overly elaborate test list—focus instead of making the first two or three iterations. This will usually get around central TDD principles like One-Step-Test, Fake-it and Triangulation, and of course the rhythm.

You *will* be required to write proper JUnit code, so be sure to train the syntax of @Test, assertThat(), @BeforeEach, etc.

One issue I see quite often is a presentation of iteration 1 (which of course is fake-it-till-you-make-it code) which then leads to a iteration 2 *that does not contribute any production code!* Other pitfalls are iterations that *produce production code not driven by the tests* or *loosing focus, writing tests for different parts of the specification.* Consult the Kata on TDD on one of the weekplans for examples.

Needless to say, this is a "failure in understanding the core of TDD", which influence the grade negatively.

2.2 Regarding Systematic black-box testing

2.2.1 Regarding the theory

The cory theory is the definition of the EC and its core properties (coverage, representation), the process of finding test cases using EC partitioning (FRS 34.2.3), and Myers heuristics for invalid and valid ECs. Also the heuristics for finding ECs (range, set, boolean heuristics) are important.

As always, just learning the definitions by heart is not that interesting, but understanding why is more so. The *representation* property is key to understanding the EC technique.

Remember also that FRS Chapter 2 contains the basic terminology of testing—it is a good idea to review what a *test case* really is.

2.2.2 Regarding solving the exercise

Be sure to read the exercise so you solve the right exercise...

Look for conditions in the specification and apply the heuristics (range, set, boolean / FRS 34.2.1) on them to get a first rough guess at the ECs. If there are many conditions, them pick just some of them, so you have time also to use the technique to generate some of the test cases using Myers two heuristics (p. 405).

Either prepare your EC and test case tables in a diagram editor, or write them on the whiteboard as you explain your reasoning.

2.3 Regarding Variability management

2.3.1 Regarding the theory

Chapter 7 / Strategy outlines the four classic techniques for handling variability: source code copy, parametric, polymorphic, and compositional, and discusses their benefits and liabilities.

It is also adviceable to have a good understanding of the 3-1-2 process and the underlying compositional design principles.

2.3.2 Regarding solving the exercise

Be sure to read the exercise so you solve the right exercise...

Look for the variability points in the exercise! What is it that you need to support in two or more variations?

Train yourself to code a given variability point in the three major code based variants: parametric, polymorphic, and compositional. For instance, code HotStone winner strategies using the three techniques.

Express your solution using code fragments written using a not-too-fuzzy code editor.

Often the exercise will ask you to provide both a compositional as well as parametric/polymorphic solution. If pressed for time, focus on the compositional one.

2.4 Regarding Test Doubles and unit/integration testing

2.4.1 Regarding the theory

The central concepts are direct and indirect input, depended-on unit, and the replacement of these, as well as the different types (stubs, fake objects, spies). Next, the levels of testing (unit, integration, and system) from FRS 8.1.5. are relevant.

2.4.2 Regarding solving the exercise

Be sure to read the exercise so you solve the right exercise…

Look for the aspects of the exercise that is not under automated test control (random values, sensor values, hardware input, system clock input, etc.)—and use the standard compositional approach for encapsulating the indirect input behind interfaces.

Remember to demonstrate your ability to express the theory in concrete code by developing code fragments/diagrams.

2.5 Regarding Design patterns

2.5.1 Regarding the theory

The central theory is that of compositional design principles as the "engine" inside almost all design patterns. In addition it is obvious to include the definitions of design patterns (there are four in the book), and at least the ones in Chapter 9.

Review the design patterns in the FRS catalogue (or you can find them all in the excellent poster by Simon Kracht / find the link on weekplan 3) and review their Intent section (you can find that in the Design Pattern Index which you can download from baerbak.com under "missing insets in new printing".)

Many patterns have a rather similar structure to STRATEGY but review closely those that are a bit different. E.g. the BUILDER has its `getResult` method only in the delegates and the instance is declared by the concrete type, not by the interface type (why?); PROXY and DECORATOR has a special structure, etc. For top presentations, these details are important.

2.5.2 Regarding solving the exercise

Be sure to read the exercise so you solve the right exercise. . .

These exercises are typically of type "identify the pattern that can solve this problem." If you do not immediately see which one it is, then do not panic. It is better to use the 3-1-2 process and compositional design principles to come up with a flexible compositional solution to the exercise and then we can probably discuss our way towards a concrete pattern during the examination.

Remember to demonstate your ability to express the theory in concrete code by developing code fragments and diagrams.

2.6 Regarding Compositional design

2.6.1 Regarding the theory

The central topics are FRS Chapter 16, 3-1-2 and the compositional design principles, and Chapter 15 about roles, interface segregation principle, and role and private interfaces. The Chapter 17 about multi-dimensional variance is also fine to relate to.

It will be natural to connect to the themes of design patterns, frameworks, and variability mangement.

2.6.2 Regarding solving the exercise

The typical exercises require you to identify variability points in the problem, and use 3-1-2 and compositional design to provide as flexible solution.

While the exercise most likely will not require any role/private interface, it is a topic that will be asked to consider from a theoretical perspective. Be sure to review your role/private interfaces usage in your mandatory project, to be able to provide examples.

2.7 Regarding Frameworks

2.7.1 Regarding the theory

Framework theory is basically the 3-1-2 process and compositional design principles but with some additional/supplementary terminology. So, be sure to include concepts like *hotspot, frozen spot, and inversion of control*, as well as understand the characteristics of frameworks. The TEMPLATE METHOD is a central pattern to understand, in both its two variants - *separation and unification.*

Time permitting in your exam planning, have a look at the source code for MiniDraw (guided by the slides and FRS 30) and review how the concepts are applied in practice.

It will be natural to connect to the themes of design patterns, compositional design principles, and variability mangement.

2.7.2 Regarding solving the exercise

The typical exercises require you to identify variability points in the problem, and use 3-1-2 and compositional design to provide as flexible framework.

2.8 Regarding Clean Code and Refactoring

2.8.1 Regarding the theory

The central theory of Clean Code is the definition and understanding of the properties outlined by Robert Martin, specied up with those that I have formulated. Bringing a Clean Code sheet to the exam may help to get an overview.

2.8.2 Regarding solving the exercise

The typical exercises contain a code section with unclean code, so solving the exercise is largely a matter of spotting the clean code properties that are not kept (document and explain it using the clean code sheet), and refactor the code into a form which is more clean.

Remember that the theme involves *refactoring*: do not just present a refactored design that you made in the preparation time. We need to understand and see your *refactoring process*—so pick one problematic area in the code base, and then demonstrate the code that replaces that particular problem, and move on to the next problematic part.

One pitfall I have seen students fall into is *writing a new method from scratch* instead of refactoring the provided code example. This is **NOT a proper solution** and will fail the exam. The focus is improving existing code!

2.9 Regarding Broker

2.9.1 Regarding the theory

The central theory of Broker is the architectural pattern, so be sure to understand UML structure, and the roles involved: what each role is responsible for, and what protocol is involved—that is—the call sequence and transformation of data along the call sequence. Also ensure to understand the typical parameters that pass between roles, as this helps you in designing solutions to the exercises—to get the method parameter lists correct.

Passing object "references" from server to client has its own set of methods and techniques that you have to master: The creation of objectId, the use of Name Services, the client having to create ClientProxies based upon returned objectId, and of course the key point—that the server can look up the proper objects using the name service.

2.9.2 Regarding solving the exercise

The typical exercises present some kind of Java interface that allows clients to call methods that are executed on a remote server—and your task is to sketch the design (typically adapting the Broker UML to the concrete exercise) and sketch parts of the Java code for those Broker roles that are not general: ClientProxies and Invokers.

All Broker exercises will involve passing server created objects.

So - be sure to study the TeleMed, the GameLobby, and your own Broker solution code base: understand which roles can be implemented in general and may thus be in a library; and which roles you always must code yourself. Also ensure you can sketch the code that handles the mechanism for transferring server created object to the client: the use of object identities, name services, and proxy creation on the client side.

While you can bring much code that you have made in the preparation, remember to carefully explain the code you show: In the ClientProxy tell us what the individual parameters are (object id, operation name, that X.class, parameters), what they represent; and similar on the Invoker side.

If you get overwhelmed by the exercise, a fall-back strategy is to sketch a design using only the "Broker I" part of Broker—that is ignore the issues of "server created objects". That way, you can ignore the issues of name services, objectId creation, etc., and focus on getting ClientProxy and Invoker code presented but treat the return value as a 'pass by value' object.

To examplify this, consider method `Card getCardInField()` from HotStone. In your presentation, you can just treat Card as an pass-by-value object that GSon can marshall and demarshall for you (just as if the return type was "String" instead).

Of course, you do not score top marks doing this, but rather a confident simple solution than a completely messy full solution.