Practical Task 5.3

(Distinction Task)

Submission deadline: 10:00am Monday, May 18
Discussion deadline: 10:00am Saturday, May 30

General Instructions

In object-oriented design, a ***pattern*** is a general repeatable solution to a commonly occurring problem. It is not a finished design that can be transformed directly into code, but rather a description or template for how to solve a problem that can be used in many different situations. Effective software design requires considering issues that may not become visible until later in the implementation. ***Design patterns*** can handle such issues and speed up development process by providing tested, proven development paradigms.

This practical task introduces you to the ***State Design Pattern*** and make you familiar with problems posed by event-driven systems. The state pattern is close to the concept of ***finite-state machines*** and allows an object to alter its behaviour as its internal state changes. It can be interpreted as a strategy pattern, which is able to switch a strategy through invocations of methods defined in the ***pattern's interface***. This pattern helps to achieve the following.

- It makes a class independent of how state-specific behaviour is implemented.
- It enables to add new states by defining new state classes.
- It allows to change the class's behaviour at run-time by changing its current state object.

Indeed, implementing state-specific behaviour directly within a class is inflexible because it commits the class to a particular behaviour and makes it impossible to add a new state or change the behaviour of an existing state later independently from (without changing) the class. The state pattern overcomes this challenge. First, it defines separate (state) objects that encapsulate state-specific behaviour for each state. This implies an ***interface*** (state) for performing state-specific behaviour. It then defines classes that implement the interface for each state. Second, it ensures that a class delegates state-specific behaviour to its ***current state object*** instead of implementing state-specific behaviour directly.

Now, let's focus on the task description. Imagine that you have been hired by an electronics company to build the software for a simple Reaction-Timer game. The Reaction-Timer machine has two inputs:

- a coin-slot that starts the game and
- a Go/Stop button that controls it.

There is also a display that indicates to the player what they should do next. The machine must behave as follows.

- Initially, the display shows 'Insert coin', and the machine waits for a player to do so.
- When the player inserts a coin, the machine displays 'Press GO!' and waits for the player to do so.
- When the player presses the Go/Stop button, the machine displays 'Wait...' for a random time between 1.0 and 2.5 seconds. After the random delay expires, the machine displays a time-value that increments every 10 milliseconds, starting at zero. The player must now press the Go/Stop button as soon as possible – the goal of the game is to show fast reactions! If the player presses the Go/Stop button during the random delay period, i.e. the player tries to "guess" when the delay will expire, the machine aborts the game and immediately demands another coin. To put it simply, there is no reward for trying to cheat! If the user has not pressed stop after two seconds, the machine will stop automatically – no living person could be that slow!

- Whether the player has pressed the Go/Stop button within the two seconds waiting time period or not, the machine displays the final timer value for three seconds, then the game is over until another coin is inserted. If the player presses the Go/Stop button while the measured reaction time is being displayed, the machine immediately displays 'Insert coin'.

1. Start with exploring the two provided templates for the program that you will need to write. You are free to select any of these two. The first option is a Windows Forms application, which is suitable for MS Visual Studio on Windows platform. The second option is a Console Application, which is cross-platform and appropriate for both MS Visual Studio and Visual Studio Code. Whichever the template you will select, your program must consist of the following parts.
   - A main program module, which is ready for you and provided in the `SimpleReactionMachine.cs` file. This module emulates the Reaction-Timer machine itself and serves as a base to test and use the controller, a part that constitutes your particular task. The controller will drive the machine.
   - A GUI component for the Reaction-Timer machine, which is also ready and provided. Its configuration depends on the template you will select. In any case, it includes a button labelled 'Coin inserted' and a button labelled 'Go/Stop'. It also has a display region to show the messages of the machine. The GUI conforms to the `IGui` interface.
   - Simple Reaction Controller is the module that you will need to implement. This is the main body of the exercise. You are free to write this component in any way you choose, so long as it implements the required `IController` interface. Specifically, your task is to add a `SimpleReactionController.cs` source code file and complete a new `SimpleReactionController` class that functions as prescribed for the Reaction-Timer machine above. The `IController` interface is included in the templates.
2. Spend some time to elaborate on how the Reaction-Timer machine as an event-driven system should act. You may wish to watch the recording of the workshop session where we demonstrated how the game works. Explore the notes on the implementation of finite-state machines (FSM), of which the Reaction-Timer machine is an example. Here, we refer to the book chapter attached to the project.

   To proceed to the next step, you should first complete the design of the system by building what is called as a state-transition diagram. The referred document has a number of very similar examples and will guide you in decision making on the number of required states and their properties as well as possible transitions. This task is crucial as the translation from the diagram to a program code is quite straightforward, sure if your diagram is correct. Certainly, many programmers are tempted to say "I can get this right without a FSM". Unfortunately, this statement is rarely true. Most often, the program that results will be hard to understand, hard to modify, and will not be correct. The document will try to convince you of the advantage of FSMs. It explains how to convert the correct design into errorless code.
3. If you trust your design and it meets the machine's specification, it is time to think about the code you will need to write. Again, we refer to the attached document and examples that it describes. Some video materials referred at the bottom of the task sheet will also help you with the structure of the correct implementation. You should know what you are going to write before you start. Focus on the use of the given `IController` interface and exploit the ***dynamic-dispatch mechanism*** that eliminates the need for the switch statement. In addition, search and read about implementation of so-called ***nested*** (also called ***inner***) ***classes*** in C# that allow to define a class within another class. This will allow you to strengthen encapsulation in your program and manipulate the states.
4. It is finally the time for coding. If you prefer to build a console application, import the files from the '`SimpleReactionMachine Console`' directory. If your choice is a Windows Forms application, then open the existing project from '`SimpleReactionMachine WinForms`', which will ask you to add the missing SimpleReactionController.cs file to the prepared project. In both cases, you will face compilation errors due the lack of the required source code file. Therefore, add the `SimpleReactionController` class and make sure that it implements the `IController` interface and contains the following methods.

- **void Connect(IGui gui, IRandom rng);**

  Connects the controller to the specified GUI component, which implements the `IGui` interface, and the specified random number generator.

- **void Init();**

  Initialises the controller.

- **void CoinInserted();**

  Reacts on the insertion of a coin into the machine.

- **void GoStopPressed();**

  Reacts on the pressing of the Go/Stop button of the machine.

- **void Tick();**

  Reacts on the **Tick** event addressed to the controller.

Obviously, your controller needs a source of timing information. You must obtain this information by implementing the `Tick` method in your controller. The main `SimpleReactionMachine` class guarantees to call this method every 10 milliseconds. Do not use any other timer. Furthermore, your controller also needs a source of random numbers. You must obtain your random numbers by calling the `GetRandom()` method of the provided generator. Do not use the `Random` class. Note that time must be displayed with two decimal places. For example, a value of 1.5 seconds must be shown as 1.50.

5. As you progress with the implementation of the `SimpleReactionController` class, you should start using the `Tester` class from the '`SimpleReactionMachine Tester`' directory of the project. It will allow you to thoroughly test the class you are developing aiming on the coverage of all potential logical issues and runtime errors. This (testing) part of the task is as important as writing the controller itself. To activate testing, make a separate Console Application project using the files in the '`SimpleReactionMachine Tester`' and import your current version of the `SimpleReactionMachine` class. Explore the `Tester` class to get details of the tests we prepared for you and the sequence that they are applied. If you fail a test, then the logic (behaviour) of your controller is incorrect.

   The following displays the expected printout produced by the attached `Tester`, specifically by its `Main` method.

```
test A: passed successfully
test B: passed successfully
test C: passed successfully
test D: passed successfully
test E: passed successfully
test F: passed successfully
test G: passed successfully
test H: passed successfully
test I: passed successfully
test J: passed successfully
test K: passed successfully
test L: passed successfully
test M: passed successfully
test N: passed successfully
test O: passed successfully
test P: passed successfully
test Q: passed successfully
test R: passed successfully
test S: passed successfully
test T: passed successfully
test U: passed successfully
test V: passed successfully
test W: passed successfully
test X: passed successfully
test Y: passed successfully
```

```
test Z: passed successfully
test a: passed successfully
test b: passed successfully
test c: passed successfully
test d: passed successfully
test e: passed successfully
test f: passed successfully
test g: passed successfully
test h: passed successfully
test i: passed successfully
test j: passed successfully
test k: passed successfully
test l: passed successfully

====================================
Summary: 38 tests passed out of 38
```

6. Finally, remember that this task is primarily about object-oriented design rather than pure coding. If you follow the instructions, your code should not be longer than (approximately) 200 lines. This is how long our solution is. Our past experience says that students coding without design will likely end up with several times longer code, which is usually messy and inefficient because of many plugs and conditional statements.

Further Notes

- Explore the attached book chapter entitled "Notes on Finite State Machines" to learn about the concepts you will need to complete this project correctly. Especially, focus on the examples of the state-transition diagrams as they will help you sketch your FSM.
- The following video materials will give you more insights on the finite state machines, their relevance to the state design pattern, and how to implement the pattern in code.
  - https://www.youtube.com/watch?v=N12L5D78MAA
  - https://www.youtube.com/watch?v=MGEx35FjBuo
  - https://www.youtube.com/watch?v=rs7DXnEmHMM
  - https://www.youtube.com/watch?v=1CAO-l6k-j0
- The following links are to cover several nuances that will likely encounter while working on this task.
  - https://www.geeksforgeeks.org/nested-classes-in-c-sharp/
  - https://www.codeproject.com/Tips/875393/Static-Dynamic-Dispatch-ReExplained
  - https://www.tutorialsteacher.com/csharp/csharp-interface
  - https://www.dotnettricks.com/learn/designpatterns/state-design-pattern-c-sharp
  - https://dotnettutorials.net/lesson/state-design-pattern/

Marking Process and Discussion

To get your task completed, you must finish the following steps strictly on time.

- Make sure that your program implements the required functionality. It must compile and have no runtime errors. Programs causing compilation or runtime errors will not be accepted as a solution. You need to test your program thoroughly before submission. Think about potential errors where your program might fail.
- Submit a picture of the state-transition diagram that you prepared as part of designing the controller.
- Submit the expected code file as an answer to the task via OnTrack submission system. Cloud students must record a short video explaining their design, work and the resulting solution to the task. Upload the video to one of accessible resources, and refer to it for the purpose of marking. You must provide a working link to the video to your marking tutor in OnTrack.

- On-campus students must meet with their marking tutor to demonstrate and discuss their solution in one of the dedicated practical sessions. Be on time with respect to the specified discussion deadline.
- Answer all additional questions that your tutor may ask you. Questions are likely to cover lecture notes, so attending (or watching) lectures should help you with this **compulsory** interview part. Please, come prepared so that the class time is used efficiently and fairly for all students in it. You should start your interview as soon as possible as if your answers are wrong, you may have to pass another interview, still before the deadline. Use available attempts properly.

Note that we will not accept your solution after the submission deadline and will not discuss it after the discussion deadline. If you fail the deadline, you also fail the task and this may impact your performance and your final grade in the unit. Unless extended for all students, the deadlines are strict to guarantee smooth and on-time work throughout the unit.

Remember that this is your responsibility to keep track of your progress in the unit that includes checking which tasks have been marked as completed in the OnTrack system by your marking tutor, and which are still to be finalised. When marking you at the end of the unit, we will solely rely on the records of the OnTrack system and feedback provided by your tutor about your overall progress and quality of your solutions.