|  |
| --- |
| **EASJ Notes** |
| Advanced Software Construction |
| Supplementary Notes & Exercises |

|  |
| --- |
| By Per Laursen  25-01-2019 |

[Introduction 2](#_Toc2095160)

[Exercises 3](#_Toc2095161)

[ASWC.0 3](#_Toc2095162)

[ASWC.1 6](#_Toc2095163)

# Introduction

These notes contain supplementary notes and exercises for use in the course “Advanced Software Construction” (ASWC).

# Exercises

|  |  |
| --- | --- |
| **Exercise** | ASWC.0 |
| **Solution** | RpgV0 |
| **Purpose** | Try to implement a piece of non-trivial business logic, with (almost) complete freedom with regards to approach. |
| **Description** | The project contains a class model for a quite simple Role-Playing Game (RPG). The main ingredients are:   * **Player**: A participant in the game. A Player will have one of three roles: Hunter, Wizard or Warrior. The most noteworthy Player fea­ture is that a Player is able to cast Spells. * **Spell**: A Spell adds a certain “benefit percentage” to any Player affect­ed by the Spell. Example: A Spell has a benefit percentage of 10 %; a Player who normally deals 30 damage points will now deal 33 damage points. Furthermore, a Spell can only affect Players of certain roles, so a Spell also contains a list of role “beneficiaries”, i.e. a list of those roles which can benefit from the Spell. * **World**: The world in which the Players act. It is initially very simple, and essentially only manages the initial setup of Players and Spells. More specifically, the setup progresses as follows:   1. A set of Spells are created, including the list of role beneficiaries for each Spell.   2. A group of Players are created, including a list of Spells that each Player may cast.   3. Each Player is now allowed to cast two spells. The specific choice of Spells casted by the group is “uncoordinated”, i.e. it is not random, but no deeper thought has gone into it w.r.t. how to optimally cast Spells (see later).   Once the Spells have been casted, the following rules apply with regards to how Spells affect individual players:   1. A Player can in principle be affected by all Spells casted by the group. 2. However, a Player will only be affected by those Spells which match the role of the Player, i.e. the role of the Player must be in the list of benefici­aries for the Spell. 3. A Player does not benefit from being affected by the same Spell more than once, i.e. the effect of identical Spells does not “stack”. 4. Spells will affect a Player in an “additive” way: Suppose a Player is affected by three Spells, with benefit percentages of 10, 25, and 20 %, respectively. The total effect on the Player will then be 10 + 25 + 20 = 55 %, resulting in a total damage percentage of 55 + 100 = 155 %. |
| **Steps** | **Step A**  The property **DamagePercentages** in the **World** class is intended to return the total damage percentage for each Player, given the current setup of the World. The property is used in the method **RunWorldExample** in **Program**. However, the property has not been correctly implemented yet – this is your job! Implement the proper­ty **Damage­Percentages** correctly, in whatever way you feel is best ☺. The expected output for the test with the given setup is (for now, ignore the line reading “Average damage percentage…”):    **Step B**  The setup of the World with regards to available Spells, participating Players and what Spells to cast is currently hard-coded into the **World** class, by means of the method **Setup**. Change this to enable the creator of the **World** object to supply setup information as well. If this involves new classes/methods and/or changes to existing classes/methods, feel free to do so.  **Step C**  In the test output from Step A, a line has been added which shows the average damage percentage for all the participating Players. Implement the functionality to calculate this average, and update **RunWorldExample** accordingly.  **Step D**  In the given setup, the spell casting was described as “uncoordinated”, meaning that each Player did not take the spell-casting of other Players into account. This will probably imply that some Spells are cast twice or more – which is wasteful, since no extra benefit is gained – while other spells are not cast at all. Since the purpose of spell-casting is to increase the overall average damage percentage, the average provides a nice way to evaluate different strategies for spell-casting. See if you can come up with more advanced strategies for coordinated spell-casting, using the average damage percentage as evaluation. Also consider:   * How robust is my strategy for changes to Players and Spells? * How much information is needed about Players and Spells in order to implement a spell-casting strategy? * Can I design the World in such a way that it becomes possible to switch between alternative strategies for spell-casting at run-time? |

|  |  |
| --- | --- |
| **Exercise** | ASWC.1 |
| **Solution** | ReportTool |
| **Purpose** | Rewrite a piece of software with the intention of improving the design w.r.t. division of responsibilities, low coupling, etc.. |
| **Description** | The project revolves around three domain classes **Customer**, **Product** and **Ship­ping­Box**. The intention is to model a scenario for e.g. a webshop, which needs to keep track of several instances of each of the three domain classes.  The project is also intended to reflect a scenario where the development of the software has been uncoordinated, resulting in several different solutions to simi­lar problems. More specifically, the project contains the class **DomainModel** (in the folder **Model**), which is supposed to store objects of the three classes men­tion­ed above. As you can see from the code, these three collections of objects are maintained in three different collection types; a **ProductCatalog** for products, a **ShippingBoxes** class for shipping boxes, and finally just a **List** for customers.  In order to keep track of the objects in the domain model, a tool for generation of a report has been created; the **ReportGenerator** class in the **Reporting** folder. The tool is functional, but has some significant design issues. The most obvious design issues are:   * The general logic for report generation is repeated three times. * The formatting of the report is hard-coded into the class.   Since the webshop anticipates that the system will soon need to manage items of other types (like e.g. **Employee** and **Order**), and will also need to be able to gene­rate reports in several formats, it has been decided to rewrite the **ReportGene­ra­tor** code. |
| **Steps** | Rewrite the code from the **ReportGenerator** class, the only hard restrictions being:   * It must still be possible to invoke the report generation functionality through an interface of type **IReportGenerator**, since other systems use that interface. * You may not make any changes to the existing classes   Apart from this, you are free to define as many new interfaces, classes, etc. as you see fit, and you may – but you are not required to – use whatever lan­gu­age constructions you wish, like e.g. LINQ. However, the primary drivers in your design should be:   * Classes should have few (ideally only one) main responsibilities. * Classes should assume as little as possible about each other. * As few classes as possible should have to be changed, if the code needs to handle new domain class and/or new report formats. |