SENG301: Practical Work

Lab 2 2022

Design principles

1. The National Squash Association (NSA) wants to build a software system to keep information about registered players. This includes details such as name & address, together with data such as club membership and tournament results.

   The NSA has three kinds of competitions. Junior players are under 16 years old — they are classified as senior players when they turn 16. Only junior players can enter junior competitions and only senior players can enter senior competitions. Players 35 or older are masters players and can enter masters competitions.
   a. How should the various kinds of players be modelled? Use plantUML to sketch the corresponding UML class diagram.
   b. Would you change your design if the NSA allowed masters players to also enter senior competitions? If so, how? If not, why not?
   c. Would you change your design if the NSA created a new kind of player, elite, who can enter international competitions? If so, how? If not, why not?
   d. Explain your designs to someone else (such as your neighbours or tutor) and discuss any differences you identify.

2. In class we discussed “single responsibility” and “interface segregation”. Refresh your understanding of these, somewhat related, concepts.
   a. Consider some of the most mission-critical classes and interfaces in your reference system. For each, write a sentence that describes, clearly and concisely, its purpose.
   b. Now decide whether there are any that don’t seem to have a single responsibility. Did you find yourself writing “and” in any of the descriptions? These may be likely candidates. Are any refactorings suggested by your conclusions?
   c. How “big” are the interfaces of these classes? We can count the number of methods in a Java interface. Similarly, we can think of the public methods of a class as its interface and count those. What happens if we include/exclude inherited methods?
   d. Are there any interfaces that should potentially be split?
   e. Discuss your conclusions with someone else.

3. We have discussed dependency inversion in class. Ideally, high-level things should not depend on low-level things: both should depend on abstractions.
   a. Consider the Car & Engine example used in class. Extend the design by adding some more classes (e.g. Truck, V8, …).
   b. Sketch some (pseudo- or Java) code to show how this would work in practice.

c. Now imagine the company wants to add electric options to the range of engines that it uses. What, if any, changes to the design (and code) would you advise?
d. Identify and discuss the design principles involved.

4. Many of the classes in java.lang.util have a method, iterator(), that returns an iterator for the corresponding collection. ArrayList is a familiar example. This allows explicit iteration over collections.

   Java also has support for implicit iteration via the "foreach" form of the for statement. For example: `for(Shape s : myShapes) { … }` could be used as an alternative to explicit iteration.
   a. Refresh your knowledge of Java's collections e.g. by consulting the corresponding javadoc at https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/Iterator.html. In particular, see how iterator() works.
   b. What happens if we delete elements during the iteration process? Do implicit and explicit iteration behave in the same way? Write some simple code to create, populate, and iterate over a collection — explicitly and implicitly. Any element type will do here — `Integer` will be fine.
   c. Now investigate what happens when you delete elements from the collection during iteration. Do both iteration types behave as you expected??

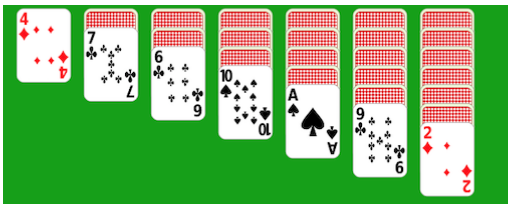
5. We have covered the Singleton design pattern in class.
   a. Are there any classes in your reference system that should be singletons? If so, then modify the code so that they are made singletons.
   b. If not, then design, implement and test a simple singleton. Think of an appropriate name for it.
   c. Document the pattern instance using the format discussed in class.

6. There are many games that can be played with a standard deck of cards. Their rules, number of players and playing procedures vary considerably, but there are some aspects that many games have in common. One of these is setting up the game by distributing — *dealing* — cards to the players before play begins.

   A standard deck consists of 52 cards. Each card has a value (1–9, Jack, Queen, King, Ace) and a suit (spades, hearts, diamonds clubs). In the game of bridge there are four players. The dealer shuffles the deck and then gives the first (top) card, face down, to the player on her left. She then continues in a clockwise direction, giving the top card to each player until they have all been distributed. At this point, each player has 13 cards in their hand and the game can begin.

Let's consider writing a Java application for playing card games — we'll start with the dealing process for a game of bridge.

a. Some (very simple) code is available to help you get started. Obtain copies of Card.java, CardPlayer.java, Rank.java & Suit.java and familiarise yourself with them.
b. Let's use the GoF Iterator design pattern to make things easier when we come to add other games to our application. Use plantuml to draw a class diagram showing how Iterator can be used to support dealing in bridge. Use stereotypes and/or notes to show where the elements of the pattern occur. Discuss your design with your neighbours and/or tutor and resolve any differences.
c. Implement your design (KISS!) together with sufficient tests to show how it works. Does the pattern seem suitable in this context?
d. In the (imaginary) game *DoubleDeal*, there are two players. The dealer gives each player the top two cards from the deck, repeating until they are all gone. Can your design be extended to include this game? Update your class diagram and code to include this.
e. In some variants of the game of *Solitaire* (*Patience*), the first 28 cards are dealt, one at a time, and placed into a triangular pattern (we need not concern ourselves with the arrangement process for now). During the game, the player (there's only one) can see the next card and choose whether or not to add it to the pattern. In this game, "next" means every third card (i.e. the 3rd, 6th, 9th etc) of those remaining in the deck. When the end of the deck is reached, dealing resumes from the beginning.



   Can your design be extended to include this game? Update your class diagram and code to include this.
f. In the game of 500 there are 4 players and a deck of 43 cards is used. The 4 twos, the 4 threes and the fours of spades and clubs are removed and a Joker is added to the deck. Would it be possible to include games such as this in your design? You need not consider the actual dealing process here.