CS3217 PS/Tutorial 2 Solution

Eric Han

January 28, 2022

Scan your attendance

Figure 1: Attendance

Plan for Today

In your groups, we will

1. 0.5 hr - [3 Groups] Chit-Chat:
    1.1 Introduce yourself
    1.2 Discuss Chp. 3
    1.3 Discuss Chp. 4
2. 1 hr - [5 Groups] Discuss the tutorial

Each group can choose 2 things from above to present.

Readings Chit-Chat - Chapter 3: Functions

Let's discuss:

- Small!
- Do One Thing
- One Level of Abstraction per Function
- Switch Statements
- Use Descriptive Names
- Function Arguments
- Have No Side Effects
- Command Query Separation
- Prefer Exceptions to Returning Error Codes
- Don't Repeat Yourself
- Structured Programming
- How Do You Write Functions Like This?

Readings Chit-Chat - Chapter 4: Comments

Let's discuss:

- Comments Do Not Make Up for Bad Code
- Explain Yourself in Code
- Good Comments
- Bad Comments

Tutorial Question 1

Suppose we are to implement a Tic-Tac-Toe game. We will need an ADT to represent the game state. Suppose we want to use TDD to implement this ADT. Think about the tests we should have for the ADT and game.

Describe the process of the development; Break the steps down into the small parts. ie. If you are the Project Manager, how would you plan it?

Tutorial Question 1 - Answer

1. Start with something small - basic or trival functionality:
   - like testing that the board is empty,
   - testing that adding marks work, etc.
2. Start going into more Tic-tac-toe specific logic, such as:
   - marking with the same marks consecutively,
   - marking the same cell,
   - marking outside the 3 by 3 grid, and so on.
3. Finally, we can focus on the winning logic:
   - check for horizontal, vertical, and diagonal.
   - We should also check that we cannot place any marks after the game ends.

Tutorial Question 2

i. What are test doubles?
ii. How do you use test doubles with dependency injection?
iii. Test this function, possibly refactoring it to be more testable (Assume that the fns. `Database.queryAll` and `EmailService.sendEmail` exist.):

```
func sendEmailToUsers(message: String) {
    let userEmails = Database.queryAll(table: "users").map { $0.email }
    EmailService.sendEmail(to: userEmails, message: message)
}
```

iv. When is dependency injection useful? When is it not so useful?

Tutorial Question 2 - Answer

Question 2i

According to the Google Testing Blog: Testing on the Toilet: Know your test doubles, test doubles are defined as follows:

- A *test double* is an object that can stand in for a real object in a test, similar to how a *stunt double* stands in for an actor in a movie

Question 2ii

Any thing that is a `singleton` must go.

Bonus Qn: Is there a good use justification for `singleton` pattern?

Question 2iii

Basically, this is how you would want to do it:

```
func sendEmailToUsers(message: String, database: PDatabase,
  emailService: PEmailService) {
    let userEmails = database.queryAll(table: "users").map { $0.email }
    emailService.sendEmail(to: userEmails, message: message)
}
```

We refactor `PDatabase` and `PEmailService` to be both protocols;

In the tests, you can mock them out in this sense:

```
// At test utils
struct DatabaseMock: PDatabase { ... }
struct EmailServiceMock: PEmailService { ... }
func testSendEmailToUsers() {
    let database = DatabaseMock()
    let emailService = EmailServiceMock()

    sendEmailToUsers(message: "hello", database: database,
        emailService: emailService)
    XCTAssertEqual(emailService.sentEmails, ["a@example.com"])
    XCTAssertEqual(emailService.sentMessage, "hello")
}
```

Question 2iv

In order to discuss the usefulness of Dependency Injection,

Advantages (etc. . .):

- Flexibility; Configurable
- Configuration is decoupled from code - improved reusability
- Able to abstract system specific configuration

Disadvantages (etc. . .):

1. Requires much, much more development time
2. Code difficult to read

Useful: Unit Testing

Not so useful: Prototype creation

Tutorial Question 3

Refactor the code snippet below to adhere to the SLAP better.

What is SLAP?

Tutorial Question 3 - Answer

```
func sha1(_ message: String) -> String {

    var digest = Digest(h0: 0x67452301, h1: 0xEFCDAB89, h2: 0x98BADCFE,
                        h3: 0x10325476, h4: 0xC3D2E1F0)

    let data = prepareData(from: message)
    for chunkIndex in stride(from: 0, to: data.count, by: 64) {
        let chunk = processChunk(from: data[chunkIndex ..< chunkIndex+64])
        let update = getNextUpdate(from: digest, with: chunk)
        digest.update(with: update)
    }

    return digest.output()
}
```

Tutorial Question 4

i. What is the delegate pattern?

ii. Suppose we want to build a user interface for the Sudoku puzzle that we have built in Problem Set 0. We will have a `SudokuGridView` UI component (that subclasses `UIView`); this component represents the grid that is displayed to the user. The user can then input numbers in this grid to attempt the puzzle.

This `SudokuGridView` contains no domain logic. Instead, it will communicate with a delegate class that we can specify with a `delegate` property in the class, i.e.

```
class SudokuGridView: UIView {
    var delegate: SudokuGridViewDelegate?
}
```

Here, `SudokuGridViewDelegate` is a protocol that acts as a delegate to`SudokuGridView`. Suggest methods that should go in this protocol.

Tutorial Question 4 - Answer

Question 4i

Idea – use object composition.

Question 4ii

Here is a sample protocol:

```
protocol SudokuGridViewDelegate {
    func sudokuGridView(_: SudokuGridView, shouldPut: Int,
        at: Cell) -> Bool
    func sudokuGridView(_: SudokuGridView, didPut: Int, at: Cell)
    func sudokuGridViewShouldUndo(_: SudokuGridView) -> Bool
    func sudokuGridViewDidUndo(_: SudokuGridView) -> Bool
}
```

Here is another possibility with different level of abstraction:

```
protocol SudokuGridViewDelegate {
    func sudokuGridView(_: SudokuGridView, colorOfCell: Cell) -> UIColor
    func sudokuGridView(_: SudokuGridView, heightOfRow: Int) -> CGFloat
}
```

Note that there are other names for this pattern, if you are a GoF (Gang of Four) follower. The refactoring.guru website has good descriptions on the patterns that are being used. Nevertheless, I would encourage less on remembering the exact names and more on recognising where they can be useful, why there are useful, and actually using them.

Tutorial Question 5 - Discussion

Watch this video on Functional Core, Imperative Shell (It's only 13 minutes long! It probably can change the way you architect software!)

Comment on the paradigm used by this video. Some guidelines:

- What are the advantages proposed by the "functional core, imperative shell" paradigm?
- In what situations might this paradigm not be applicable?
- The video mentions that only the functional core needs to be tested; the imperative shell needs no tests. What do you think about this?

Read more: Boundaries in Practice

Tutorial Question 5 - Discussion

Watch this video on Functional Core, Imperative Shell (It's only 13 minutes long! It probably can change the way you architect software!)

Comment on the paradigm used by this video. Some guidelines:

- What are the advantages proposed by the "functional core, imperative shell" paradigm?
- In what situations might this paradigm not be applicable?
- The video mentions that only the functional core needs to be tested; the imperative shell needs no tests. What do you think about this?

Read more: Boundaries in Practice

- Pure functions: always return the same outputs for inputs; no side effects
- How can we implement hash tables in say Haskell?
- What can go wrong?