475 Software Engineering for Industry: Topic 1 : Legacy Code

Part I:

Legacy code refers to the code written by predecessors in a broad sense. Compared with non-legacy code, the code is usually lacking some corresponding tests[1]. Most software developers are reluctant to process and modify strange legacy code. However, many companies still have a number of legacy codes. The following will explain why so many companies have legacy code, although no-one likes to use it.

- Everything is evolving and changing, including software. The Software will automatically switch to maintenance mode if the version control stops at some point. Once the software enters the maintenance phase, it becomes legacy [4].
- In the development of some projects, it is difficult to discard legacy code directly, especially to develop systems with quite complicated business logic, such as some financial telecommunications systems. Obviously, the legacy code can complete the system function throughout the business time. If we abandon it directly, redevelopment will cost a lot of resources and is not necessarily successful. If the new requirements change, there is often no time to redevelop the code.
- Experiencing with legacy code is unavoided for every developer. Everyone is capable of working in a fresh environment where everything is new while using legacy code is quite a difficult task. However, writing legacy code can hit some important deadline due to a range of business reasons [5].
- From the perspective of the company's cost, using legacy code can sometimes yield better returns. Money spent on a quick-and-dirty project that allows immediate entry into the market may be better spent than money spent on designing better structure [2]. As long as the initial release is below the design pay-off line introduced in the design Stamina Hypothesis. Because sometimes it is better to trade off design quality for speed [6].
- In addition, there is often a phenomenon of employee turnover in the workplace. Once the person responsible for the software being developed leaves, the rest of the code naturally becomes legacy code.

Reference:

1. Michael Feathers, Working Effectively with Legacy Code, Upper Saddle River, N.J.: Prentice Hall PTR 2004
2. Foote, Yoder, Big Ball of Mud, University of Illinois at Urbana-Champaign 1304 W. Springfield Urbana, IL 61801 USA
3. https://simpleprogrammer.com/deal-with-legacy-code/
4. https://blog.rinatussenov.com/what-is-legacy-code-is-it-good-for-you-fb260a467fb7
5. https://martinfowler.com/bliki/TechnicalDebt.html
6. https://martinfowler.com/bliki/DesignStaminaHypothesis.html

Part II:

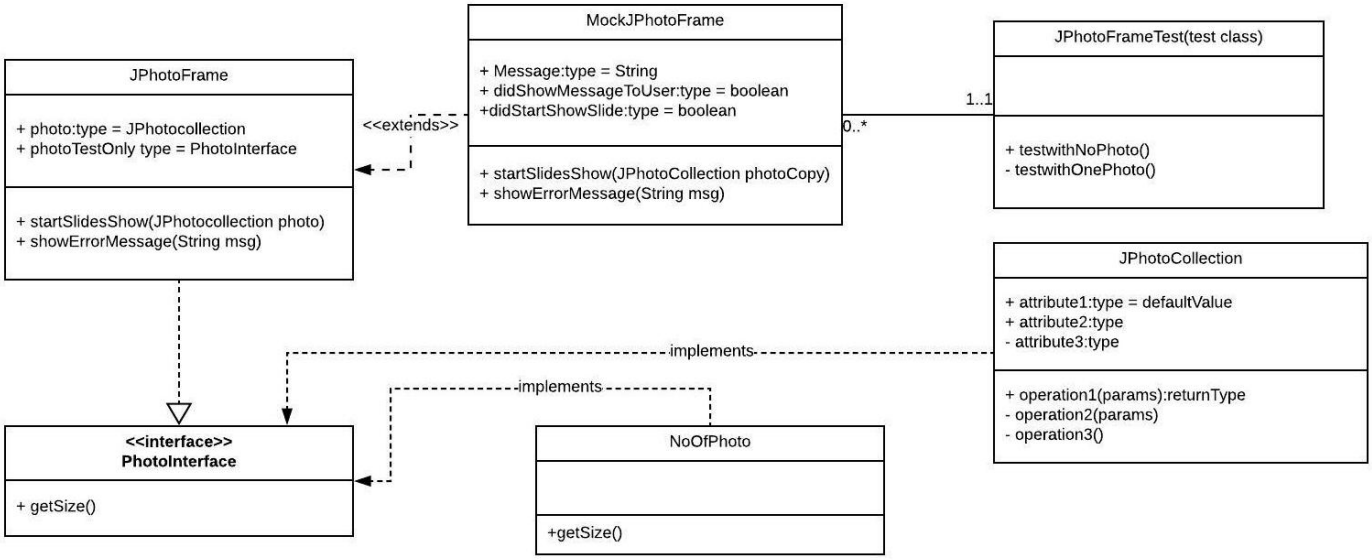

Figure 1: UML

To perform unit test for "Start Slideshow" button, we extract the related code block from `actionPerformed()` method in `JPhotoFrame` class, so that some mock objects can call that part of code without executing other parts. Afterwards, in `JPhotoFrameTest` class we create a subclass `TestingJPhotoFrame` that extends `JPhotoFrame` class, to instantiate some mock objects. The subclass has some overridden methods which has different utility from its super class, i.e., instead of display slides (if size of photo album > 0) or prompt error message (if size of photo album $= 0$), methods in subclass can be overwritten to recording some actual value that help us testing by assserting to some expected value.
We perform two test cases to ensure that wo possibilities after clicking is both correct. In the first test case, we have a abstract frame (Not real Photo frame window) which contains no photos and test whether it will show correct error message with expected `"No photos to show!"` The second one we have another abstract frame which has exactly one abstract photo in album and test whether it will show the slide (it will not show the slide in real, but returns a `boolean` value to pretending it is showing). More logically, instead of using the `MockJPhotoFrame` in the test cases we can break dependency by instanciating `NoOfPhoto`, so that we can get rid of calling other part in `JPhotoCollection`,

```
// testing "Start Slide show" feature
else if (cmd.equals(JPhotoMenu.A_SLIDESHOW)) {
    showPhotoSlides(photos);
}
void showPhotoSlides(JPhotoCollection photoCopy) {
    if (photoCopy.getSize()>0) {
        startSlidesShow(photoCopy);
    }
    else {
        showErrMessage("No photos to show!");
    }
}

public void startSlidesShow(JPhotoCollection photos2) {
    JPhotoShow show = new JPhotoShow(photos2, 5000, list);
    show.setVisible(true);
}

public void showErrMessage(String msg) {
    JOptionPane.showMessageDialog(this, msg,
            APP_NAME, JOptionPane.ERROR_MESSAGE);
}
```

```
class MockJPhotoFrame extends JPhotoFrame {

    public String Message;
    public boolean didShowMessageToUser;
    public boolean didStartShowSlide;

    protected MockJPhotoFrame() throws Exception {
        super();
    }

    @Override
    public void showErrMessage(String msg) {
        didShowMessageToUser = true;
        Message = msg;
    }

    @Override
    public void startSlidesShow(JPhotoCollection photoCopy) {
        this.didStartShowSlide = true;

    }
}
```

```
public class JPhotoFrameTest {
    @Test
    public void testWithNoPhoto() throws Exception {
        MockJPhotoFrame testOne = new MockJPhotoFrame();
        JPhotoCollection empty = new JPhotoCollection();

        testOne.showPhotoSlides(empty);

        assertTrue(testOne.didShowMessageToUser);
        assertEquals("No photos to show!", testOne.Message );
    }

    @Test
    public void testWithOnePhoto() throws Exception {
        MockJPhotoFrame testTwo = new MockJPhotoFrame();
        JPhotoCollection onePhoto = new JPhotoCollection(new String[] { "photo" });
        testTwo.showPhotoSlides(onePhoto);
        assertTrue(testTwo.didStartShowSlide);
    }
}
```

Figure 2: Extracting behaviour out; Subclass for testing; Unit test class conducting 2 test cases

Testing things are all passed in as parameters so that this code is hard to test. In terms of tradeoffs, instead of using PhotoInterface we instantiate the JPhotoFrame and pass some range string to make the photo size > 0 or we can instantiate the JPhotoFrame by using the default constructor. We make this tradeoff because instantiating the JPhotoFrame does not have a lot of external dependencies and is less time-consuming. By using the interface in a large-scale project, we might introduce an anonymous bug into the preious code so that it becomes more time-consuming.
Testing driven design may change the code structure because a common intuition for the developer is to satisfy the testing rather than implementing good structure.

To add new feature as "Quick Slideshow" we add another button in `JPhotoMenu` class, as well as another event (`elseif` block) under `actionPerformed()` method in `JPhotoFrame` class, where we perform similar implementation as original Slideshow apart from changing parameters passed into `JPhotoShow` from `5000` to `50`, which means reducing time intervals from 5000ms to 50ms. Similar unit tests can be applied and details can be found on GitLab:
https://gitlab.doc.ic.ac.uk/jz2618/jphotoalbum.git