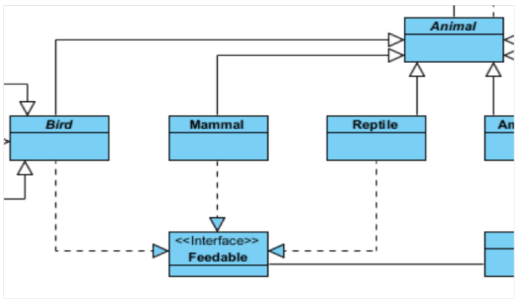
Task1

Relationship between software maintenance and writing maintainable code

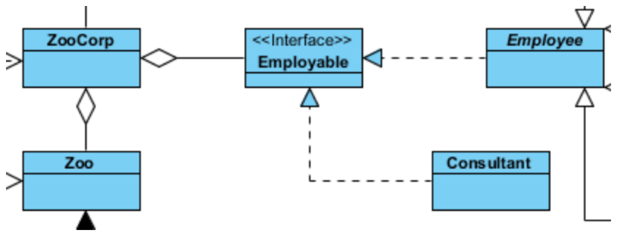
Software maintenance and writing maintainable software are intimately connected, regarding their mutual benefits. Both of them should not be realized with an afterthought but cautious approaches as they are vital to the whole project development. On one hand, writing maintainable software needs to take responsibility for relieving the stress during period of software maintenance. Several tasks, such as making codes complying with conventions, refactoring massive methods or classes and even rearchitecting with hierarchies, could be done in advance [1]. Alternatively, software maintenance is concentrated on faults correction, increasing adaptability to changeable environment and performance enhancement [2]. It could provide some high-level advice, such as preventive maintenance for those who is writing maintainable software.

Illustration with example

Let's take the zoo project as an example. An abstract super class, "Animal", is designed to be extended over shared figurative subclasses, such as "Bird", "Mammal" and "Reptile". In this way, a potential big class is broken down into independent units, which increases the modularity and analyzability. Among them, an interface called "Feedable" is designed for methods override, which avoid unnecessary code repetition, making things explicit and convenient extension.

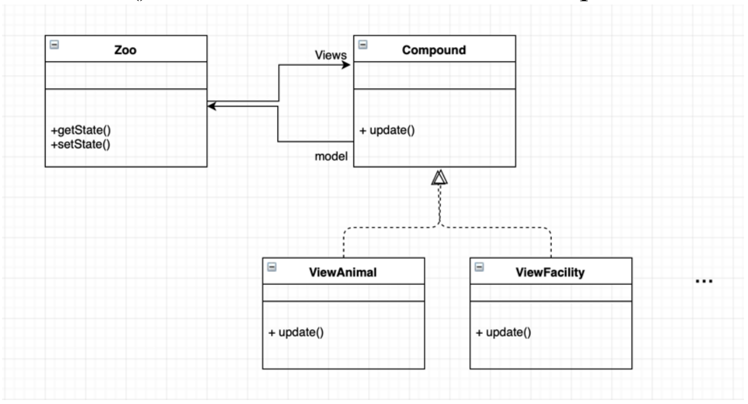


Besides, apparent hierarchy is set up with utilizing aggregation and composition. Explicit layer architecture is shown: "ZooCorp", "Zoo" and "Animal". Among them, interclass dependency has been instantiated: an aggregation between "ZooCorp" and "Zoo" to show that "Zoo" is a part of "ZooCorp" and the latter one could easily be extended with extra details, like "Employee". In this way, "ZooCorp" can be easily modified even deleted without influencing stuff in "Zoo". Thus, codes of different classes are less likely to influence each other and thus, become more maintainable.

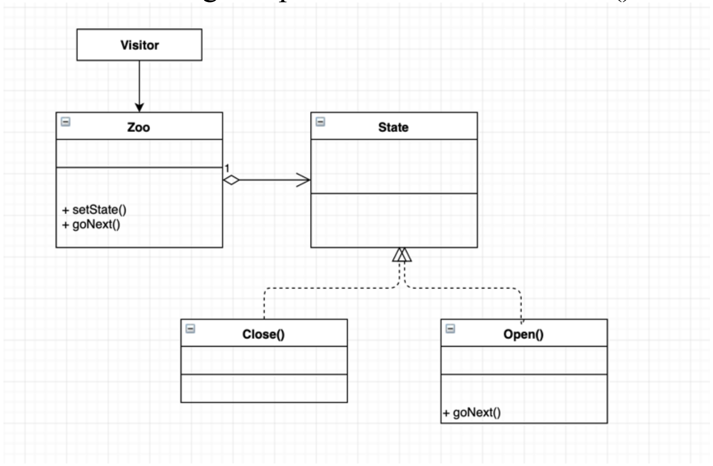


Design Pattern

Furthermore, common core solution to the project could be described by design patterns. For the zoo project, two design patterns could be applied: observer and state. Observer can be used between "Zoo" and "Compound" to interpret one-to-many dependencies. Once changes are made in "Zoo", attributes and fields of "Compound" will be adapted to its latest modification. As the following picture shows, one "Zoo" could have several "Compounds" and view their states. Then, encapsulate "getState()" and "setState()" in "Zoo" and variables in "Compound".



Besides, state design pattern could be used between "Zoo" and "Visitor" as "Zoo" can decide whether to open or not regarding the number of visitors. Firstly, create an abstract state class and detailed state behavior. Then, make a pointer to the current state "Open". If there are too many visitors that beyond restriction number for the purpose of safety, "Zoo" will change its pointer to the state "Close()".



Robustness

However, the above zoo project is not perfect yet as more robustness could be added. For example, as mentioned before, the utilization of above design patterns can increase structural robustness as they promote reusability and make it easier to understand codes and future debugging.

Furthermore, refactoring methods could be used to provide greater consistency for user since they improve internal structure without altering external behaviors. Adding regression testing and reducing coupling also makes the software more robustness

Task2

Process of regression testing

Through the task in lab5, the process of regression testing could be roughly divided into several small parts. Firstly, unit-level tests for “Employee”, such as check salary and bonus calculation methods, are conducted to confirm that their original functionalities work as expected [3]. In this way, it is helpful to expose integration issues on early stages rather than leaving the potential problems that may influence additional tests. After that, entry point and entry criteria are identified to assure codes has satisfied preset functionality. This kind of tests are designed where assuring the “animal” arraylist is empty for follow-up invertebrate classes establishment. In addition, tests are executed to ensure stability and sanity of current codes since new classes with internal methods are just implemented [4]. This is done by adding two spiders to “Compound” class and running the previous-passed tests again to verify whether previous functionality is stable or not. Moreover, a large number of test cases including 10 million “Fish” instances are designed to test the boundary and to define exit criteria using statement “timeout==1000”.

Debugging vs Testing

Debugging is the process of faults identification, analysis and correction, whose premise is that there exist errors in the software [5]. Conversely, testing more concentrates on bug-free software verification and finding potential problems which will not influence compilation but the requirements. During the lab, we first write codes and debug to make sure there aren’t fatal and apparent errors. After that, tests are carried out to analyze performance of each aspect and to see if all the requirements are correctly satisfied. Additionally, nearly no design knowledge is needed by testing but extremely required for debugging since errors often take place when implementation logic is wrong. This may also contribute to the reason why testing could be done automatically but debugging must be manual. For example, a large number of test cases including 10 million “Fish” instances are designed to execute automatically. However, this cannot work on debugging.

Importance of regression testing for legacy software

A legacy software is the program which has been around a long time. Since business requirements are continually explored, it often needs to be updated. However, changes of implementation or environment high likely to cause potential problems and it is difficult work with legacy codes. Regression testing is valuable since it ensures that the entire software works to expectation with adding new features. With regression testing, the software will timely detect errors and correct them without leaving the hidden troubles which may lead to huge business loss. As a result, future maintenance becomes easier and stability increment.

Test-driven development (TDD) and regression testing

TDD is the process that designers use minimum number of codes to pass the test defining new improvement in advanced [6]. TDD could be utilized to build up testing scaffolds which could be used in regression testing.
The core of TDD is using testing to actuate future code implementation followed by refactoring. Moreover, codes are proven to meet the requirements. Conversely, regression testing is often put out during or after adding unproven feature into the software.
These techniques not only make legacy codes maintenance simpler but convenient, by using existing scaffolds and framework. Furthermore, they increase the effectiveness by isolating the code being testing with its surroundings. Also, refactoring helps with breaking up legacy codes and add new characteristics.

Reference

[1]. Farrell, J. 2001. JAVAWORLD: Make bad code good. [Online]. [3 November 2019]. Available from: https://www.javaworld.com/article/2075129/make-bad-code-good.html
[2]. Joy, S. Unknown. 2012, GeeksforGeeks. [Online]. [3 November 2019]. Available from: https://www.geeksforgeeks.org/software-engineering-software-maintenance/
[3]. Software testing. Unknown, Software Testing. [Online]. [3 November 2019]. Available from: http://softwaretestingfundamentals.com/smoke-testing/
[4]. Ranorex. Unknown, Ranorex. [Online]. [3 November 2019]. Available from: https://www.ranorex.com/resources/testing-wiki/regression-testing/?creative=304774075802&keyword=&matchtype=b&network=g&device=c&utm_source=google&utm_medium=cpc&utm_term=&utm_campaign=uk_irl_en_dsa&gclid=EAIaIQobChMIjISIy5vJ5QIVxbHtCh0KbQgdEAAYASAAEgJiP_D_BwE
[5]. Geeksforgeeks. Unknown, GeeksforGeeks. [Online]. [3 November 2019]. Available from: https://www.geeksforgeeks.org/differences-between-testing-and-debugging/
[6]. Farcic, V. 2013. Technology conversations. [Online]. [3 November 2019]. Available from: https://technologyconversations.com/2013/12/20/test-driven-development-tdd-example-walkthrough/