Software Engineering

Unit 1 : Answers

Submitted by -

Team 5
Ranjith Raj Vasam - 201460624
Ankita Kumari - 201101161
Avinash Chandwani - 201350882
Somay Jain - 201101054
Shivani Poddar - 201156152

Answer 1:

The design of any system is always performed in iteration. It cannot happen in a single attempt. The initial design forms the baseline of ideas and requirement upon which multiple levels of things and reviews are performed with different views and perspective of seeing the problem and on multiple attempts, the design can be improved and optimized. Also the initial design gives us confidence as well as **baseline upon which we can work for continuous improvement**. The second reason is there is always requirement for change during the course of designing systems, so if we have an initial design, we can try out for change on top of it as well as alternative approaches of design can also be applied keeping the initial design as baseline. Here the initial design, means the system at high level, should necessarily **meet the business requirement**. Finally, a good design **incorporates space for extensibility** at time of requirement for newer modules.

Answer 2:

In most of the scenarios, the pair or peer programming model reduces the need for refactoring the code, separately after development. Here the model defines that a pair of developers work together, while one writes the code, other observes, guides or reviews the code continuously, hence the **code smells often caught** at the earliest place. Hence, in most of scenarios the peer programming model often **saves on effort needed on refactoring the code later** after development or detailed review by the peer.

A real time example would be suppose two developers are working following peer programming model, a developer is writing a piece of code to test certain functionality is working fine, here at this moment other developer can see the piece of code as possibly been called at several places, hence 2nd developer can pin-point this as reusable code, which can be moved inside a separate method, hence this refactoring won't require testing later. Hence this model reduces a lot on refactoring the code.

It also helps in incorporating multiple perspectives into the design and the subsequent implementation. For instance, while working on the design problems in class, all the members of the group at times catch lazy namespaces so as to come up with a collaborative generalized and informative one. Thus, accentuating on the merits of peer programming.

Answer 3:

This statement in the famous paper "No Silver Bullet -- Essence and Accidents of Software Engineering" by Fred Brooks seems to be accurate for several reasons.

First we distinguish between the two categories of complexity in software development: the essence and the accident. The essential complexity of software development is related to the specification, design (mapping specification to software), and testing (that the design properly meets business needs). The accidental complexity refers to the difficulties related to implementation (languages, runtime, tools, and programming techniques). In his article, Brooks explained how the various innovations that attempted to address accidental complexity (at the time) were not silver bullets, and concluded that the only things that may yield close to an order of magnitude productivity improvement are those that address essential complexity, such as improvements in requirements gathering, rapid prototyping, and cultivating good design skills.

Why is that? Brooks claims this because every software has the following constraints or features:

- Software's complexity being among the most complex thing people create and increases non-linear with its size. The extreme complexity causes challenges of communication, verifying all scenarios, usage, adding new features without unwanted side effects, and security concerns.
- Conformity is a necessary quality of software for a variety of reasons, which include user interaction, interfacing to existing software, or extending existing functionality. Conformity can not simply be designed out of software, but design consideration can reduce the complexity of conformity challenges.
- Changeability is listed as an essential difficulty for software development, because all successful software gets changed, customers change their requirements, bugs are fixed, and software is infinitely malleable.
- Invisibility is the last essential difficulty that Brooks lists. Geometric abstracts, blueprints, flow charts, and models aid in visualization of software. Creating, reading, and communicating these visualization aids add complexity and opportunity for errors.

However, Brooks wasn't right about everything. It must be noted that some of his claims about the future of software development were inaccurate. In particular, his claims about high-level language advances, object-oriented programming, environments/tools, and workstations have all been proven to be false.

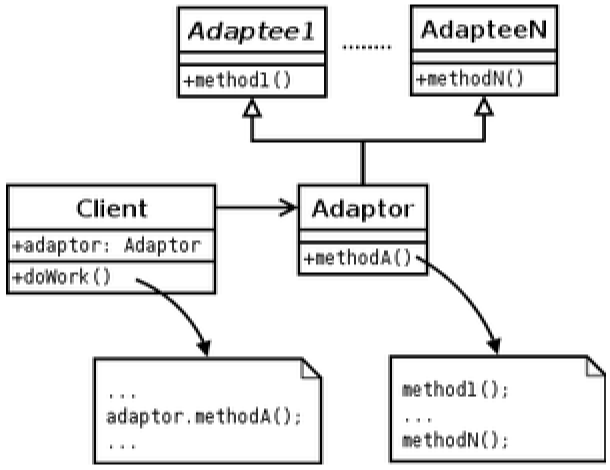
In conclusion, there is no denying that Brooks had a lot of valid points in this paper backed by solid proof and justifications. But one must keep in mind the era in which this article was published. In the late 1980's software development wasn't as advanced as it is today. For example, the concept of internet, object-oriented programming etc were not as easily accessible as it is today. This is why in the No Silver Bullet Reloaded Retrospective OOPSLA Panel Summary, Fred Brook himself claims that "I would question whether there has been any single technique that has done(or showed potential to become a Silver Bullet) so since(the paper was published). If so (i.e. if such a silver bullet existed), I would guess it is object oriented programming, as a matter of fact."

Answer 4:

"An adapter is a glue class, but not a wrapper class". It serves as a proxy between otherwise incompatible parts of software, to make them compatible. The standard practice is to keep logic out of the glue code and leave that to the code blocks it connects to.
For example In Java Programming language Integer is a wrapper class, it doesn't act as adapter, it encapsulates the primitive int data type. Hence an adapter pattern is a glue class and not a wrapper class.

Answer 5

The adapter pattern is used in java uses multiple polymorphic interfaces to implementing or inheriting both the interface that is expected and the interface that is pre-existing. It is typical for the expected interface to be created as a pure interface class



Answer 6:

Definition of iteration logic - Embedded iterators mix structure maintenance and traversal and allow only one traversal at a time. Also, because they are present in the collections class, adding different traversal algorithms makes it clumsy. On the other hand, separate iterators are independent of the collections. They allow multiple traversals at the same time and support various traversal algorithms on the same collections.

Location of control - External iterators allow the client to do the iteration using the provided methods. However, for an aggregate, the client would have to recursively iterate through the collection on her own. In cases like these, internal iterators are very useful. The iterator itself controls the traversal. The client simply hands the iterator some operation to be performed on each element in the aggregate.

Answer 7:

In order to give external iterators privileged access to a collection's data members in C++ you would make the iterator a friend using the 'friend' keyword. In Java privileged access is granted by using the package-private. Internal iterators do not need to use any special constructs in order to gain privileged access.

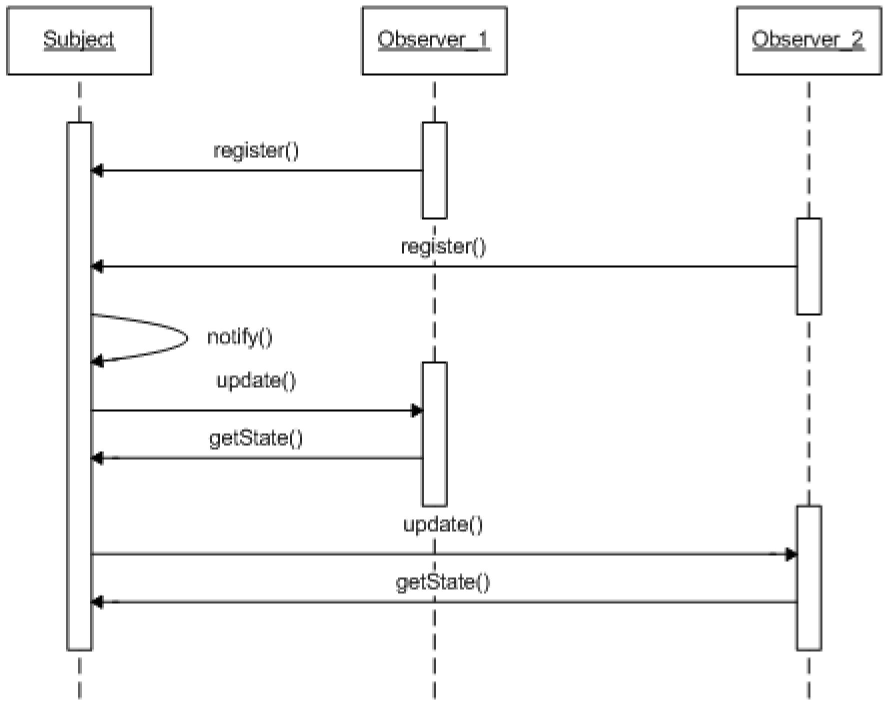
Answer 8:

It is difficult to iterate using an external iterator because any position in the structure may span many levels of nested aggregates. So, the external iterator needs to store the path traversed so far through the composite to keep track of the current object. For eg, in case of a binary tree, we would have to store the object at each step we took (left/right) so that we can return to the next node on fully traversing the nested tree. It is similar to writing a iterative traversal for a recursive data structure.

If at all we need to do this, we can maintain a stack of the objects encountered so far. When we get hasNext()==null for the current object, we remove it from the stack.

Answer 9:

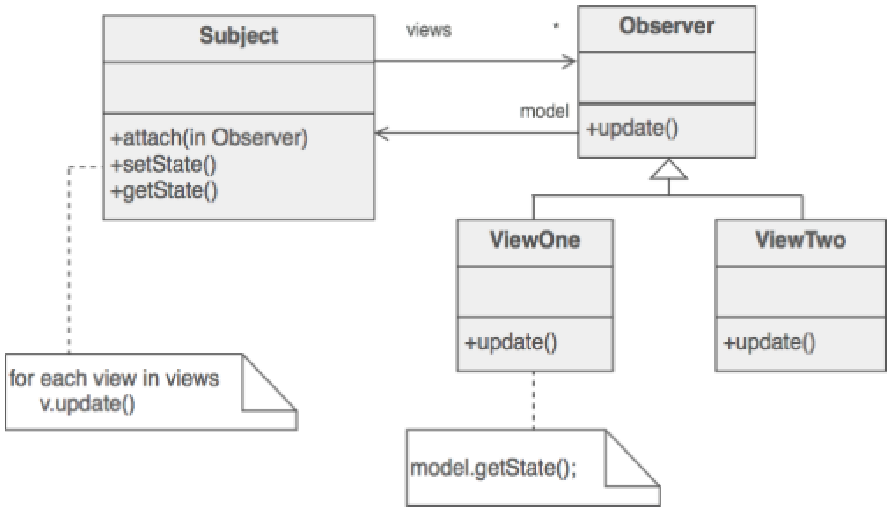
The Observer pattern decouples the Observer from the Subject and it is more efficient because the Observer can be automatically notified of any pertinent change, rather than the Observer having to continually ask the Subject if a change has occurred. This means that the Observer and the Subject only interact when it is absolutely necessary (i.e. when the Observer registers itself and when the Observer needs to be notified of a change).



Answer 10:

Swing is a heavy user of the Observer pattern—it implements more than 50 event listeners for implementing application-specific behavior, from reacting to a pressed button to vetoing a window close event for an internal frame. GUI is the foremost module which interacts with the user and mediates the input from the user to the controller and the output from the code to the user. Thus, it is utmost essential that the GUI be lag free and updates the respective component as soon as anything is changed. For instance, when we query for zoom of an image in a website, if the GUI takes long to process the same notification of "updating" the older image to its zoomed version, the user will not be able to see the same. Thus, although the changes would be made in the backend, they would not reflect timely in the front end for the user to experience the same.

This needs to be handled at the design level itself so that the programmers find it easier to inculcate in the final code. This should be ideally handled by the framework itself. For instance in the MVC architectures, this is usually taken care of. The Observer pattern also on the same lines has one subject and potentially many observers. Observers register with the subject, which notifies the observers when events occur. The prototypical Observer example is a graphical user interface (GUI) that simultaneously displays two views of a single model; the views register with the model, and when the model changes, it notifies the views, which update accordingly. Thus, it should be incorporated at the framework level of any application. It would be thus essential in eliminating the concern for synchronous notifications to the GUI as discussed earlier.



Answer 11:

The disadvantages of including the functions to handle the composite's children in the component interface was at first unclear, but after some discussion, we determined that too much generalization could be a result. (Discussed as a group with all members present)

The functions to handle the composite's children should be included in the component interface because it makes it easier to add new components; doing this can make handling composites simpler because there is no need to have knowledge at runtime that it is a composite, rather than a leaf. Adversely, adding these functions to the component interface can make the system too general, thus making it more difficult to add type restrictions to the composite later (they would have to be runtime checks).

Answer 12:

The major limitations in building large software have always been (a) its brittleness when confronted by problems that were not foreseen by its builders, and (by the amount of manpower required. The recent history of expert systems, for example highlights how constricting the brittleness and knowledge acquisition bottlenecks are. Moreover, standard software methodology (e.g., working from a detailed "spec") has proven of little use in AI, a field which by definition tackles ill- structured problems. How can these bottlenecks be widened? Attractive, elegant answers have included machine learning, automatic programming, and natural language understanding. But decades of work on such systems have convinced us that each of these approaches has difficulty "scaling up" for want a substantial base of real world knowledge.

Current software tools supporting diagramming notations are not satisfactory. Editors for diagramming notations such as state transition networks, Petri nets or the entity-relationship data model, are always confronted with a problem: how much guidance should be given to the user throughout the editing task? Not enough guidance allows the diagram to evolve to nonplausible configurations and may provoke the user to feel lost in the editing process. At the other extreme, if too much guidance is provided the user feels like being shepherded through the diagram drawing; this results in an obtrusive and unfriendly system. Current tools normally offer a trade-off solution based on the introduction of some semantic constraints in the diagram editor to forbid a number of operations. To assert the correctness of the diagram, the reset must explicitly request it to be checked. I believe this solution is not satisfactory. All the semantic constraints should be embedded in the editor in order to allow automatic diagram validation. The challenge is: how to do it without limiting the user's freedom during the editing task? I propose an approach that provides a solution to this problem

Typically there is an abstract Builder class that defines an operation for each component that a Manager may ask it to create. The operations do nothing by default. At the implementational level, the issue of Assembly and Construction Interface where builders create their product in step by step fashion, this builder class must be general enough to allow construction of products for all kinds of components. A model of construction and assembly where the results of construction requests is just to append to the product is usually sufficient. But when it comes to situations like the one given, you might need access to parts of the product that are constructed earlier. e.g. Tree structures such as parse trees which are built bottom up. In this case, the builder would return child nodes to the Manager, which then would pass them back to the builder to build the parent nodes. This design mechanism now can enforce semantic constraints on the tree construction. Z is formal specification language used for semantic constraints check.

Example:

An efficient spatial data structure in a GIS system for database updating is required in order to minimising of spatial constraint
violations and timesaving. An automated constraint checking procedure has been introduced to perform constraint violations check
at compiling time before updating the database. Formal definitions of spatial data types were used in attempt to formulate novel
equations and architectures to detect constraint violations and framework for spatial repository. A data structure called Semantic
Spatial Outlier R-Tree (SSRO-Tree) was proposed for the Semantic Integrity Constraints Checking System to improve the
functionality of the proposed method. The R-Tree or its variants have been widely-used data structures for this purpose and which
are based on a heuristic optimization but unable to perform semantic spatial join queries at database updating. An experiment was
conducted using actual spatial data and results revealed that the performance of SSRO-Tree is notably superior to the R*-Tree and
RO-Tree for conducting semantic spatial join queries.

Answer 13:

On adding a new part in the product, a new method should be introduced in the Builder so that the Director is able to use it. Moreover, the Director should have proper error handling mechanisms to ensure that it does not build a part which is not present in the builder being used. So, If an older ConcreteBuilder is in use and the Director requests for a new part type not known by the old builder to be put in the product, the Director should throw an appropriate error message.