Infosys®

Building Tomorrow's Enterprise

Infosys Global Agile Developer Certification

Requirements, Architecture and Design

Study Material

www.infosys.com

Revision History

Version	Date	Remarks
1.0	Dec’2014	Initial version

Table of Contents

List of Table

1 Introduction

Requirement elicitation and prioritization is one of the most critical activities in software development. It has been observed that issues like incomplete and unclear requirement often lead to incorrect software delivery and may not provide the expected value.

However Agile methods such as Scrum and XP are based on the concept of completing small but valuable requirements in short iterations. This provides lot of flexibility to evolve and prioritize the requirements throughout the duration of software development. The developers and testers need to be part of the Project initiation and Release initiation stage for understanding requirements right from the beginning of the Agile project.

System architecture provides high-level structure to serve as a starting point for design and development activities for a project. Architecture and Infrastructure planning (AIP), and high-level organization of components at start of project can help discover and prioritize major architecture and deployment issues that require further investigation.

In Agile, the immediate focus remains on implementing a sub-set of features at any given time. The architecture and design keeps evolving along with the software as and when the new/additional features keep getting added. There is no dedicated design phase in Agile projects. The team should get aligned to the evolving architecture and design of the project. With each iteration, the Team improves the design of the system. It is the approach towards keeping the design of the system as simple, clean, and expressive as possible at all times.

The design documents are typically not prepared or maintained in Agile projects. However, it is recommended to maintain just enough documentation for the complex modules. The code is self-explanatory and forms the basis of design in Agile projects. The detail design evolves iteratively with addition of new features. The system design is kept open and simple in order to add new features.

This document elaborates on:

- Techniques for writing Requirements in Agile projects
- Requirements categorization and Prioritization in Agile context
- Architecture and Design in Agile projects
- Applying Design Principles and Overview of Design Patterns

2 Requirements

In Agile framework, entire set of requirements is broken into smaller pieces of work which can be completed in short cycles. The requirements are mostly defined in format of stories which are written from the perspective of end user.

The initial requirements can be even a one-liner. However, it doesn't mean that the requirements are unclear when team is developing the software. The Scrum Master and the entire team gets clarity in a session of interactive conversation with Product Owner (PO) during Release or Sprint level planning meetings. To get requirements in shape, they are groomed regularly and refined on ongoing basis. In the Agile project, Requirements gradually evolve with the progress of the project.

2.1 Requirement Categorization: Theme, Epic, User Story

In Agile execution, requirements or work items are classified at different levels. The factors considered for this classification are scope of requirement, dependency to deliver in cluster, and effort. In addition, enhancement in workflow for existing feature is also recommended to be written in terms of user stories to maintain consistency. A defect (deviation from requirement) is also sometimes a change request or a new requirement in the garb of bug. If identified, it should be converted in a story.

The widely-used technique for classifying the requirement is size-based SMC (Small, Medium and Complex) or based on various factors like dependency, unknowns, risk and uncertainty.

In Agile, requirements are primarily defined based on value to business (ROI-Return on Investment) and how they evolve over the project execution timespan. Requirements are broadly classified as Theme, Epic or User Story. Following is the brief definitions with some examples.

Theme

A Theme is very large requirement and may, sometimes, encompass entire product vision. Most of the times, a theme is a set of interrelated requirements which can be considered for Release planning. A large theme can be subdivided in multiple sub-themes. Development of a theme can span from few months to a year.

Example of a Theme: *'Xyz Bank wants to allow customers transact online and be able to do fund transfer, open fixed deposits, and request check book online.'*

Epic

An Epic, is very large requirement that can't be completed in a short iteration (Sprints), and hence, needs to be broken down into User stories. An Epic can be considered as a small theme by virtue of its magnitude and sometimes, is talked about as one and the same.

Example of Epic: *'Xyz Bank wants to allow funds transfer between own accounts in different branch, other accounts in same bank, and preferably, accounts in other banks.'*

User Story

A User Story is a feature small enough to be delivered in iteration and represents unique business value. Generally, an Epic is divided into multiple user stories. It's mostly a vertical slice cutting across all phases of Software development life cycle and has tangible business value. A user story can also represent technical debt which doesn't represent business value directly but is required for better and maintainable code quality.

Example of a User Story:

- *'As a customer, I want to schedule automatic fund transfer among my accounts.'*
- *'As a customer, I would like to transfer funds and get an SMS on successful transfer.'*

2.2 Requirements Prioritization

Requirements prioritization is the most important aspect to deliver high value requirements faster. The prioritization is a value-driven approach and forces business stakeholders to define requirements based on the changing business needs. According to Jim Johnson of the Standish Group, when requirements are specified early in the lifecycle, 80% of the functionality is relatively unwanted by the users. He reports that 45% of features are never used, 19% are rarely used, and 16% are sometimes used. In short, only 20% of all delivered software features are “often or always used.”

This explains why requirement prioritization is very crucial to deliver higher Return on Investment in Agile development where team is delivering software in short iterations periodically. It is important to note that the business stakeholders have the ultimate authority to decide the priority of the requirement. The prioritization techniques would help them to tune the priority based on the factors that they are not aware of or might have overlooked.

2.3 Effective Requirement Definition

When defining requirement in Product Backlog, Product Owner can take liberty of not defining the detailed requirement, in fact it can be even a single line. However, as the requirement moves from PB (Product Backlog) to SB (Sprint Backlog), it should be detailed enough for team to estimate size and effort, and break down to task. Agile does not mean no documentation, rather it is about creating useful documentation which helps in delivering right requirements.

One of the key practices used to validate requirements understanding empirically is, to have prototypes or UI (User Interface) wireframes of the workflows attached to stories and looping Product Owner to avoid rework. Another key practice is to identify dependent requirements early. Ensure that, as far as possible, the requirements are platform-independent. This would provide flexibility for implementing them.

There are Functional as well as Non-Functional Requirements (NFR). NFRs enforce some constraints and define boundaries of acceptable behavior of the system such as expecting system to respond to user action in definite time, or perform in certain way under defined user load and stress. Quite often, this important aspect to capture NFR as part of user story is missed and could result in software which is working but not meeting expectations.

2.3.1 Requirement Workshops

Requirement workshops are conducted for gathering, understanding and prioritizing requirements, mainly during the Initiation phase (also called as Sprint-0 or Discovery phase). The outcome of these workshops is Epics and Stories on which stakeholders have enough clarity. These workshops have stakeholders from even extended teams such as Tech Architect, Solution Designer, IT Managers, and BAs (Business Analysts) to ensure teams are collectively aware of what they intend to complete in that Release. In this workshop, requirements are prioritized, detailed out, estimated. Also, rough timelines are decided for Sprint and Release based on business needs.

For more guidelines on conducting requirement workshops, please refer to ‘InfyAgile Global Requirement Workshop Guidelines’ document in PRidE.

Path:‘Sparsh→Webapps→PRidE→Processes→IT-Services→Engineering-Processes→Development→Agile Methodology→InfyAgile Global’

2.3.2 Requirement Accuracy

Requirement or work items should be detailed enough for team to pick up and complete. Evolving requirement is part of Agile framework and hence, it is not mandatory to have all the requirements defined to every minor detail.

However, the requirements that are taken up for development in a Sprint, should have enough clarity for estimation. Complex Epic requirement should be broken down to sub-epics till it is possible to define multiple short feature/story for each one.

The requirement becomes accurate when it is detailed into smaller stories as it will have lesser unknowns, dependencies and associated risks. Adding testable acceptance criteria in each requirement defined by business stakeholders, that clearly outline what is expected at the completion of the requirement, will help in a major way to increase requirement clarity.

2.3.3 Techniques & Practices

User Story is one of the most popular and recommended techniques for writing requirements. Product Backlog contains multiple stories, epics and is prioritized as per business value. A User Story is written in perspective of user and how it would help business. It must always capture AT (Acceptance Test) which should be testable and complete.

Sample User Story Template

<table>
<tr><td>User Story Id: UX-14
Requirement Id:</td><td>Description: Search feature in Control Panel</td></tr>
<tr><td>As a (User Role) Control Panel user
I want to:
Search all the values in the Control Panel based on Product code, Channel Code, Location, Stability Profile combination
So that :
I can filter the different values for further updates</td><td>References:
Control Panel Page
Acceptance Criteria:
- Verification test to be performed from Control Panel Page
- Verify that the correct values are displayed based on the filter criteria applied during search.</td></tr>
</table>

Story DONE Criterion (Definition of Done)

Each story should have DONE criterion to consider the story for acceptance. A simple DONE criterion can be as given below, however it can be fine-tuned based on team maturity.

- Story is developed and Unit-tested
- Peer reviewed and passes all unit tests
- QA (Testing) complete and no defect open
- Acceptance Criteria satisfied

Even though User Stories provide good enough way to capture requirement, the format prohibits getting details of all user interaction with sample data. This is when the techniques such as storyboarding or scenario sketching are used. Here entire workflow of user interaction is captured along with sample data.

For example: Following story illustrates how a scenario sketch looks like and provides starting point for creating stories, features and epics.

Hotel Booking telemarketing executive Anthony books hotel rooms for customer who calls at helpdesk. In the booking process, He receives details of customer such as email id, phone no., room preference, and enters them in the system. Based on the dates on which the customer is checking in, Anthony searches for available options

and based on confirmation from customer, blocks the rooms. The payment process must be completed in the system after which the booking will be confirmed. On confirmation, system generates booking reference number and sends email to the customer with a copy of the same also in the confirmed booking DB (database).

2.3.4 Three 'C's For Writing Good User Stories

Ron Jeffries, the co-inventor of the User Story practice and founder of XP (eXtreme Programming), refers to three important components of user story: Card, Conversations and Confirmations, otherwise known as "The Three C's". The terminology 'Card' came from XP world where practice of writing requirement on Index cards has flourished. The purpose of using cards for writing requirements was to keep it short and leave room for other 'C', (Conversation) to get more details about the story.

Card provides a starting point for defining requirement and conversation brings required details. There may be more than one conversations needed to get enough clarity for the team. If the team is following cards, only the conversation summary must be recorded on the backside of the card with date.

The third 'C' is Confirmation i.e. Acceptance criteria. This criteria provide ways for development team to confirm if they are meeting the story card needs. Sometimes these can prove as excellent artifacts to be considered as part of documentation deliverables.

Most of the time, in GDM (Global Delivery Model), where teams are working from multiple locations, teams use Application lifecycle management tools such as Rally, Mingle, VersionOne, Jira, rather than physical index cards. In such cases, details of conversation, discussion in planning or pre-planning should be added in story details section with date.

For additional guidelines on writing good stories, please refer to *'InfyAgileGlobal Guidelines for Writing User Stories'* document in PRidE

Path: 'Sparsh→Webapps→PRidE→Processes→IT-Services→Engineering Processes→Development→Agile Methodology→InfyAgile Global'

2.3.5 INVEST Technique

Bill Wake, author of *'eXtreme Programming Explored'* has suggested the INVEST model which is widely used as a guideline to define stories in Agile world. It is a simple technique to assess whether or not a User Story is well-formed. Below are the details:

- **Independent:** Stories should be as independent as possible. If there are dependencies between stories, it might lead to them not getting delivered due to other story. It also adds to problems while estimating the stories.
- **Negotiable:** User stories provide brief description of the functional details which should be negotiated in a conversation between client and development team.
- **Valuable to purchasers or Users:** User stories should avoid including user interface assumption and technology assumption.
- **Estimable:** Developers should be able to estimate the amount of time a story will take to code/develop.
- **Small:** Stories should be small and concise. They should be easy to understand and estimate. It is better to have more number of short stories rather than having one large story.
- **Testable:** Stories are not contractual obligations. The agreements should be documented by tests that demonstrate the correct development of a user story. Successfully passing the tests proves that a story has been successfully developed.

2.4 Requirements Prioritization Techniques

2.4.1 Prioritizing By Value

In a typical enterprise application, there is a portfolio of hundreds of IT projects getting executed at the same time. Many of these IT projects are legacy applications running for years while few are created to address new market needs. For such large enterprise applications, realizing the business value can be very challenging and would often require a group of Product Owners or Product Managers determining the value in terms of tangible (first mover advantage, unique offering) and intangible (performance, workflow, usability improvements) benefits.

Ensure that large requirements are broken down in releasable features to realize the business value incrementally. The factors that should be considered are value stream visualization, cost to build the feature vs. efforts, feasibility, infrastructure cost (if any), risk, skills, segment/audience to which it is targeted and their needs and also whether it would make it in the MMF (Minimum Marketable Feature) list. Some requirements are essentially to improve operations or for statutory compliance reasons and do not add business value, rather they are cost realization.

2.4.2 MMF (Minimum Marketable Feature)

The term MMF comes from *'Software By Numbers'* by Denne, Cleland-Huang. In the Kanban world, the work priority or the most important requirement are talked in terms of MMF. With Lean becoming popular in recent times, MMF was reoriented in terms of Minimum Viable Product (MVP), Minimum Viable Feature (MVF) or Minimum Marketable Release (MMR).

The term 'Viable' instead of 'Marketable' is used because, for a major release of an enterprise, business value is not always based on market forces but also on what's feasible and if team is on the right track to achieve the differentiator. Similarly, MMR is clearly focused on the set of features that needs to be clubbed and released together to gain business value.

By choosing to prioritize based on MMF or MVP, team focuses on completing smallest functionality of real importance to business and end-users, rather than releasing a theme or epic which may not directly lead to delivering most important need.

For example, two banks want to launch an ATM center in a new area. To gain first mover and recall benefits, the ATM software has facility to transfer funds to credit cards. This will not fall under MMF. Rather, the most basic selling point for the ATM will be how fast bank can launch the ATM with cash disbursement.

2.4.3 MoSCoW

MoSCoW was first coined in CASE (Computer-aided Software Engineering) Method Fast-Track which was a RAD (Rapid Application Development) approach. Later, it became part of Dynamic Systems Development Method (DSDM) Consortium. It is a helpful technique to determine which requirement should be delivered first and is based on categorizing requirements based on attributes as given below.

Must Have: 'Must Have' are the requirements that cannot be negotiated and if not delivered, delivery would be considered failure. 'Must Have' requirements should fall under MMF for sure. It is important to note that business stakeholders and vendors may falsely believe everything as MUST, typically, in Fixed Price (FP) projects but it probably is just a perception and not reality.

Should Have: 'Should Have' are the requirements which are important, but not vital. Though this is a critical requirement, it can still be addressed with some workaround.

Could Have: Often, they are the nice-to-have features that generally create intangible benefits such as Client satisfaction rather than increasing real business volume.

Won't Have: Least priority at that point in time.

For example, for user login functionality, allowing user with valid credentials to proceed would be a Must, locking user account after 3 continuous unsuccessful attempts would be a Should-have feature, whereas providing a client side validation for blank inputs would fall under Could-have category of features. Similarly a login password with very stringent password formation requirements can be something that may not be the most important (Won't Have) thing for a web site which is not doing any financial transactions.

2.4.4 Cost of Delay (COD)

COD is simply the impact of not completing a user story or feature to business. It can be used with Kanban method to effectively prioritize requirements that have more value over others because of cost of non-completion.

COD or Opportunity Cost, is many times represented in graphical form as a line showing the relationship between impacts vs. time. Apparently, steeper the angle of the line, worse is the impact over time. Impact of COD can be losing first mover advantage (competition risk), compliance risk *(for e.g. banking application has to comply with policy changes which come with timeline)* or identity crisis in crowded market without the differentiator.

There are 4 profiles for COD:

- Emergency or high business impact and high risk prone
- Fixed Schedule (where, if you deliver after schedule, you incur heavy cost. For e.g. statutory or policy change)
- General or Standard (impact with medium risk)
- Investment where business is investing to make the product better (a wish list item)

2.4.5 Certainty

It is a well-known fact that certainty helps in selecting requirements. Uncertainty about end user needs or methodical solution will put the same on backburner. Certainty helps in fast-tracking the decision process of business stakeholders, whereas uncertainties possess a major risk on moving ahead even if the requirement is of high business value. Hence, a certain requirement has more weightage while prioritizing the same.

2.4.6 Feasibility

Even if requirements are critical, team needs to ensure that they understand feasibility of the requirement. Feasibility can be gauged based on the cost/benefit analysis (CBA) method as well as understanding technical complexity. No matter which technique is used, list of project requirements must be sorted from most to least valuable.

3 Architecture and Design

High-level architecture generally involves depicting the system into modules, representing the interfaces and interactions between these interfaces. Detail design outlines the design internals for each of the modules. Typically, for OOAD (Object Oriented Analysis and Design) project, high-level design includes system architecture with details about interfaces, components, and networks that need to be developed along with the data flows and interactions between component systems. Detail design further divides high-level design into class diagram with all the methods and relation between the classes. Design documents in traditional software development are very elaborate and detailed and run into several pages since the entire system design is done up-front and the design team passes on the design documents to the development team.

One of the important Agile manifesto principle stresses on good design in Agile projects "Continuous attention to technical excellence and good design enhances agility". Therefore lots of attention needs to be provided to avoid the bad design. Following are the factors which lead to bad design:

Factors	Description
Rigidity	The design is difficult to change
Fragility	The design is easy to break
Immobility	The design is difficult to reuse
Viscosity	It is difficult to do the right thing
Needless complexity	Overdesign
Needless repetition	Mouse abuse
Opacity	Disorganized expression

Table 1: Factors for Bad Design

Projects executed in traditional methodology have requirement phase where the entire system requirements are gathered up front. This is followed by the design phase where system design is detailed out and designed. Typically system architecture, interaction diagram, component model, class diagram, sequence diagram, system context diagram, use case diagram are done during the design phases (High Level/Low Level Design phases). This detailing is done before start of the coding phase and is called Big Design Up Front as all the design is complete before the start of coding phase.

Agile believes in "*Welcome changing requirements, even late in development*". As a result, Agile projects believe in *"No Big Design Up Front"* **(**NBDUF) and *"You Aren't Gonna Need it"* (YAGNI) principles. YAGNI is one of the engineering principles of Extreme Programming (XP) which focuses on working on the features which are deemed necessary. It is usually used in combination with other Extreme Programming practices of Refactoring and Test Driven Development constraints on adding new features

Rationale for YAGNI/NBDUF approach is:

1) Any code or design which is not required for the current set of features might not be required in the future
2) More time is spent in designing/developing feature which does not add any value to the current set of features required by the customer
3) Additional unrequired feature can lead to software becoming larger and complicated which in turn can impose

Product and platform should be decided at start of the project. Create visual architecture and infrastructure (AIP) planning diagram defining components, interfaces and other characteristics of system or components, as required. Any specific security or other NFRs can be outlined in the visual AIP along with product and platform being used. System Architecture is typically decided during project workshop or project initiation (Sprint 0) phase. The architecture defined takes care of the organizational IT roadmap and the application/product being developed. The following guidelines can be followed to create stable software architecture:

- Create System architecture diagram to show how system will be implemented in different layers and the main components being used
- Decide on platform level details (programming language, database)
- Identify risk areas that need POC (Proof of Concept)

Techniques used for defining system architecture are:

- Expert Judgment
- Whiteboard, AIP diagram
- Existing Enterprise Guidelines & Constraints

3.1 Define Architecture Stories

The software Architecture is the structure which comprise the software components, the externally visible properties of those components, and the relationships among the components. It represents the structure of the data and program components that are required to build a computer-based system.

Architectural Style is a transformation that is imposed on the design of an entire system. The software that is built, exhibit one of many architectural styles. Each style describes a system category that encompasses:

- A set of component types that perform a function required by the system
- A set of connectors (subroutine call, remote procedure call, data stream, socket) that enable communication, coordination, and cooperation among components
- Constraints that define how components can be integrated to form the system
- A topological layout of the components indicating their runtime interrelationships
- Semantic models that enable a designer to understand the overall properties of a system

Software architecture is refined at the initiation of the project. Risk area identified as part of the 'Define System Architecture' phase (i.e. Sprint-0 or Discovery Phase). These are defined as architecture stories and taken up as POC (Proof of Concept) during this phase, which is in-line with the idea of fail fast of Agile principle. Any other POC as required for the identified risk areas or new technology implementation should be covered during this phase and has to be in agreement with all stakeholders. Some of the high priority stories can be identified for implementation, during first 2-3 Sprints (assuming 2 weeks iteration for project duration of more than 6 months) which can set up the technical foundation (e.g. Data Access Framework, Error Logging Framework, common utilities) for the system being developed. It is important to note that the team should have good visibility of the system requirements to implement these technical stories.

High-level objectives of this phase are:

- Reduce project risk due to technology by validating assumptions (e.g. POC for identified risk areas) and identifying the dependencies between the components
- Provide foundation for future Sprints, like define error handling framework, define data services and design patterns (Can be implemented at later Sprints if the customer priority is different)

Techniques used are:

- Expert Judgment
- Team discussions
- Whiteboard

The overall quality of a software architecture can be assessed by considering the dependencies between components within the architecture. These dependencies are driven by the data (information) and control flow within a system.

3.2 Just-in-Time Design

Minimal design discussion can happen at start of the Sprint for stories for the current Sprint to ensure consistent implementation. Further focus on feature implementation and design should be given at start of story. The design typically evolves iteratively along the Sprints. The design focus during the Sprint shifts to just-in-time design. Design at the start of the Sprint sets the direction for the team for implementing the stories in conjunction to the defined architecture. Whiteboard with expert judgment is the most used tool and technique for just-in-time (JIT) design.

Typically the team during this phase has design discussion for implementation of the features. As new features are implemented, design needs to be constantly kept simple to ensure that it is both maintainable and extensible. Design guidelines, as described in the concepts section, should be applied while developing the system to ensure the same. TDD can be followed to ensure easier code refactoring during subsequent Sprints. Pair programming can be followed in the project to ensure code is simple and self-explanatory.

During the JIT design phase, certain visual artifacts as mentioned below can be created for implementing the current story at hand.

- Interaction Diagram
- ER (Entity Relationship) diagram
- Sequence Diagram

Typically, the design techniques used for JIT are –

- Expert Judgment
- Whiteboard

During Sprint phase architecture, design and code quality should be frequently monitored. It is recommended to have a code quality tool integrated with the developers IDE to ensure code quality compliance. Practices like continuous integration should be followed and any technical debt accrued should be addressed in a timely and effective manner.

3.3 Design Principles

In traditional software development project, requirements and plans are set before development begins. This enables the team to know the road ahead with a reasonable confidence that requirements or design will not change drastically in the middle. Whereas, Agile method is to enable and embrace change. On an Agile project, requirements can change at any point in the project cycle. Therefore it is imperative for Agile teams to have a code in a position where they can conveniently accept a new requirement or change. Following guidelines can be used for design in Agile projects:

1) Keep It Simple

2) Apply SOLID Principles

3) Continuous Code Refactoring

4) Test Driven Development (TDD)

5) Implement Design Patterns

Note: Continuous Code Refactoring and Test Driven Development is covered in detail in the separate study material

3.3.1 Keep It Simple

One of the key concepts in Agile development is to keep the system design as simple as possible. Simple design is adaptive in nature and easily responds to frequent changes due to changing requirements in Agile projects. Following criteria can be applied to keep the system design simple:

- **Self-Explanatory**: A code should be self-documenting. The code should use unabbreviated words i.e. full names. Standard naming convention should be followed. Use comments or assertions for better understanding of code.

- **No duplication**: Each and every declaration of the behavior should appear *OnceAndOnlyOnce*. Developers should not implement unrelated ideas in the same method. Code without duplication is easier to maintain and change.

- **Remove superfluous content**: Code should not have any unnecessary content. It should only have the code to support the required functionality. Any functionality which is not required should not be coded (YAGNI principle).

- **Cover all test scenarios**: Write code that satisfies unit tests covering all functional areas. TDD helps the developer with the same. For the required feature, minimal coding is done to keep it simple and flexible to maintain.

3.3.2 Apply SOLID Principles

SOLID is an acronym coined for five object-oriented design principles. These principles help in creating a robust system for easier maintenance. These are OOPS concepts but can be applied for any other design principles as well. Some of the SOLID principles can be debatable and should be evaluated before applying in project specific scenarios. The SOLID principles are as follows:

a) Single Responsibility Principle: As described by Robert Martin "*A class should have one, and only one, reason to change*." This principle relates to cohesion, which is a measure of how closely two things are related. The programmer should maximize cohesion so that each class does only one thing which ensures that behavior of a class is not altered accidently.

In the below code, the Single Responsibility Principle (SRP) is being violated because the 'InsertRecord' class has multiple responsibilities of creating database connection and inserting the record.

```
// Method with multiple responsibilities – violating SRP

public void InsertRecord(Shopper e)
{
        string StrConnectionString = "";
        SqlConnection objCon = new SqlConnection(StrConnectionString);
        SqlParameter[] SomeParameters=null;
        SqlCommand objCommand = new SqlCommand("InsertQuery", objCon);
        objCommand.Parameters.AddRange(SomeParameters);
        ObjCommand.ExecuteNonQuery();
}
```

The above design if not very efficient as it does not allow reuse. In the below code, the above code is re-factored to implement SRP to increase reuse. Though generally this principle is widely accepted, one of the arguments against SRP is that depending on the future requirements the code can be refactored for reuse.

```
// Method with single responsibility – follows SRP

public void InsertRecord(Shopper e)
{
        SqlConnection objCon = GetConnection();
        SqlParameter[] SomeParameters=GetParameters();
        SqlCommand ObjCommand = GetSQLCommand(objCon,"InsertQuery",SomeParameters);
        ObjCommand.ExecuteNonQuery();
}

private SqlCommand GetSQLCommand(SqlConnection objCon, string InsertQuery, SqlParameter[] Param)
{
        SqlCommand objCommand = new SqlCommand(InsertQuery, objCon);
        objCommand.Parameters.AddRange(Param);
        return objCommand;
}

private SqlParameter[] GetParameters()
{
        //Create Paramter array from values
}

private SqlConnection GetConnection()
{
        string StrConnectionString = "";
        return new SqlConnection(StrConnString);
}
```

b) Open/Closed Principle: The open/closed principle states "*software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification*"; that is, such an entity can allow its behavior to be modified without altering its source code. *Source: http://en.wikipedia.org/wiki/Open/closed_principle*

There would be various instances where the existing code will need to change the class behavior. Open/closed principle suggests extending or changing the behavior of the class without changing the class. This can be done by applying the OOPS (Object Oriented Programming) concept of abstractions and polymorphism resulting in derived class to extend the behavior for new feature.

In many practical situations with the requirements changing drastically it does not make sense to have 7 layers of hierarchy in the code because modifications were asked 7 times. The design decision should be based on how frequently the base class is being referenced. This principle is extremely useful in frameworks, libraries where there are numerous dependencies and any change to the behavior of the existing classes would cause big impact on the system.

The below code is refactored to implement Open Closed Principle to calculate the area of rectangle and circle by inheriting the base class. This is helpful as for any new change in requirements (say calculating the area of a cylinder) in the future can be handled without changing the existing code or classes making the design more responsive to changes and leaving no room for introducing any bugs in the existing functionality.

```
//Open for modification but closed for extension

public double CalculateArea(object[] shapes)
{
        double area = 0;
        foreach (var shape in shapes)
        {
        if (shape is Rectangle)
```

```
        {
                Rectangle rectangle = (Rectangle)shape;
                area += rectangle.Width * rectangle.Height;
        }
        else
        {
                Circle circle = (Circle)shape;
                area += circle.Radius * circle.Radius * Math.PI;
        }
        }
        return area;
}
```

The above code is refactored to implement Open Closed Principle.

```
public abstract class GetDimensionsForShape
{
        public abstract double CalculateArea();
}

//Inheriting from GetDimensionsForShape the Rectangle and Circle classes now looks like this:

public class Rectangle : GetDimensionsForShape
{
        public double Width { get; set; }
        public double Height { get; set; }
        public override double CalculateArea()
        {
                return Width*Height;
        }
}
public class Circle : GetDimensionsForShape
{
        public double Radius { get; set; }
        public override double CalculateArea() {
        return Radius*Radius*Math.PI;
        }
}
```

c) Liskov Substitution Principle: "*Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.*" — Robert Martin.
If a system is using a base class, the reference to that base class can be replaced with the derived class without changing the functionality of the system.
Liskov Substitution Principle (LSP) has certain violations, most typical being the Circle-ellipse problem (refer http://en.wikipedia.org/wiki/Circle-ellipse_problem for details).

The below code violates Liskov Substitution Principle. Since only 'ProjectTask' raises an exception when the task status is closed, the program needs to be changed if 'ProjectTask' is used as substitution for 'TaskStatus'.

```
public class TaskStatus
{
        public Status Task_Status { get; set; }
        public virtual void CloseTask()
        {
                Task_Status = Task_Status.Closed;
```

```
        }
}
public ProjectTask : TaskStatus
{
        public override void CloseTask()
        {
                if (TaskStatus == TaskStatus.Started)
                throw new Exception("Cannot close a started Project Task");
                base.CloseTask();
        }
}
```

The above code is modified below to support the Liskov Substitution Principle. By stipulating that a call of TaskClose() is valid only in the state when CanCloseTask() returns true, we can fix the Liskov Substitution Principle violation by applying the pre-condition to ProjectTask and TaskStatus.

```
public class TaskStatus
{
        public Status Task_Status { get; set; }
        public virtual bool CanCloseTask()
        {
                return true;
        }
        public virtual void CloseTask()
        {
        Task_Status = Task_Status.Closed;
        }
}
public ProjectTask : TaskStatus
{
        public override bool CanCloseTask()
        {
                return Task_Status != Task_Status.Started;
        }
        public override void CloseTask()
        {
        if (Task_Status == Task_Status.Started)
        throw new Exception("Cannot close a started Project Task");
        base.CloseTask();
        }
}
```

d) Interface Segregation Principle: This principle states that clients should not depend on the interface that they don't use. Each interface should have a single responsibility and should not be overloaded. Interfaces should be kept small and cohesive and exposed only when required.

```
interface IGetBankCustomerData
{
        public Customer GetBankCustomer(Guid AccountID);
        public void SetBankCustomer(Customer customer);
        public Customer GetBankCustomerForReport(Guid AccountID);
}
```

The above interface is polluted by having the third method declaration when a report data access class implements this. The report class needs only the Get Customer For Report method but for the other two methods it will have implement something, at least it will throw Not Implemented exception for the first method. So, in this case we have to segregate this interface into two, which is in-line with interface Segregation Principle.

```
interface IGetBankCustomerData : IBankReportDataAccess
{
        public Customer GetBankCustomer(Guid AccountID);
        public void SetBankCustomer(Customer customer);
}
interface IBankReportDataAccess
{
        public Customer GetCustomerForReport(Guid AccountID);
}
```

e) **Dependency Inversion Principle**: Robert Martin quotes that

- *High level modules should not depend upon low level modules. Both should depend upon abstractions*
- *Abstractions should not depend upon details. Details should depend upon abstractions*

If a class has dependencies on other classes, it should rely on the dependencies' interfaces rather than their concrete types. The idea is to isolate our class behind a boundary formed by the abstractions it depends upon. If all the details behind those abstractions change, then our class is still safe. This helps keep coupling low and makes our design easier to change. The following example considers a code to track the user actions on a website.

```
public class LogUserActions
{
        public void AddLog()
        {
        // Adds user actions to a file
        }
}
public class LogManager
{
        LogUserActions log = new LogUserActions ();
        public LogManager(LogUserActions log)
        {
                this.log = log;
        }
        public void Log()
        {
                this.log.AddLog();
        }
}
```

In case, there is a change to log the user actions in database as well then new low level class needs to be written to inset the data to the database as below. Issue with this approach is the need for change to LogManager class which will require the entire functionality to be tested again.

```
public class LogUserActionsInDB
{
        public void AddLog()
        {
                // Insert the user activity details in Database
        }
}
```

Using the dependency Inversion principle (DIP), we can put across abstract class or interface between high level and low level classes. This example is rewritten below using DIP by creating an abstraction *(IManageLog)* and LogManager as high level class.

```
public interface IManageLog
{
        void AddLog();
}
public class LogManager
{
        IManageLog log;
public LogManager(IManageLog log)
{
        this.log = log;
}
public void Insert()
{
        this.log.AddLog();
}
}
public class LogUserActionsInDB: IManageLog
{
        public void AddLog()
        {
                // Insert the user activity details in Database
        }
}
public class LogUserActions: IManageLog
{
        public void AddLog()
        {
                // Adds user actions to a file
        }
}
```

With this solution, we can implement new low level classes without changing the code for the high level class. The high level class does not depend on the low level classes, thus achieving greater flexibility. Again the implementation of this principle should not be applied to all classes. If class functionality is not likely to change then this principle need not be applied.

Note: It would be a mistake to unconditionally conform to a principle just because it is a principle. The principles are there to help us eliminate bad smells. They are not to be liberally scattered all over the system. Over-conformance to the principles leads to the design smell of needless complexity.

3.3.3 Implement Design Patterns

Experienced Object Oriented (OO) developers (and other software developers) build up a collection of both general principles and idiomatic solutions that guide them in the creation of software. These principles and idioms, if codified in a structured format describing the problem and solution and named, can be called as patterns. This pattern (also termed as Design Pattern) names, abstracts, and identifies the key aspects of a common design structure that makes it useful for creating a reusable object-oriented design.

In OO design, a pattern is a named description of a problem and solution that can be applied, to new contexts. Stated in another way, a design pattern describes a design structure that solves a particular design problem within a specific context and amid forces that may have an impact on the manner in which the pattern is applied and used. It systematically uses, explains, evaluates an important and recurring design in OO systems. It helps to

- Document design decisions and rationale
- Reuse wisdom and experience of master practitioners

- Reuse successful design
- Design alternatives
- Get right design faster

For example, here is a sample pattern:
Pattern Name: Information Expert
Problem: What is a basic principle by which responsibilities are assigned to objects?
Solution: Assign a responsibility to the class that has the information needed to fulfill it

A good pattern is a named and well-known problem/solution pair that can be applied in new contexts, with advice on how to apply it in novel situations and discussion of its trade-offs, implementations, variations and so forth. The intent of each design pattern is to provide a description that enables a designer to determine:

- Whether the pattern is applicable to the current work
- Whether the pattern can be reused
- Whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern

For more details on Design Patterns, please refer following books:

- Craig Larman, *'Applying UML Patterns : An Intro to Object-Oriented Analysis'*, Pearson Education
- Pressman, R.S., *'Software Engineering : A Practitioner's Approach'*, 7th Edition, Tata McGraw Hill

4 Practice Questions

Question 1

In Agile projects, how are the non-functional Requirements captured?

a) As a Theme
b) As a separate User Story
c) As a Task
d) Depends upon the Product Owner

Question 2

'Certainty' and 'Feasibility' are related to which of the following techniques?

a) Keep it Simple
b) Design Pattern
c) Requirement Prioritization
d) No Big Design Up Front

Question 3

Identify the Design principle from the below options which is not related to SOLID principles?

a) Simple Responsibility Principle
b) Open Closed Principle
c) Liskov Substitution Principle
d) Interface Segregation Principle
e) Dependency Inversion Principle

Question 4

Which of the following is not an essential part for describing a Design Pattern?

a) Problem
b) Pattern Name
c) Solution
d) Benefit

Question 5

Continuous Code Refactoring and Test Driven Development helps the Agile project Team to keep the design simple.

a) True
b) False

5 References

- Craig Larman, *'Applying UML Patterns : An Intro to Object-Oriented Analysis'*, Pearson Education
- Pressman, R.S., *'Software Engineering : A Practitioner's Approach'*, 7th Edition, Tata McGraw Hill
- Robert Martin *'Agile Software Development, Principles, Patterns and Practices'*
- Websites
  - 3 'C's of XP:
    http://xprogramming.com/xpmag/expCardConversationConfirmation
  - Stories, Epics & Themes from Mike Cohn:
    http://www.mountaingoatsoftware.com/blog/stories-epics-and-themes
    http://agilebench.com/blog/the-product-backlog-for-agile-teams
  - Very Detailed Paper on techniques of Requirement prioritization**:**
    http://www.ijric.org/volumes/Vol1/2Vol1.pdf
  - Minimal Marketable Features:
    http://www.mountaingoatsoftware.com/blog/make-the-product-backlog-deep
    http://www.romanpichler.com/blog/product-backlog/making-the-product-backlog-deep/
    http://www.boost.co.nz/blog/agile/story-mapping-prioritisation/
  - Requirement uncertainty:
    http://www.sei.cmu.edu/library/assets/whitepapers/nps-2010-acquisition-research-campbell-grady-paper.pdf
  - Other Nice Reads:
    http://www.poppendieck.com/pdfs/Extra%20Features.pdf
    http://www.mountaingoatsoftware.com/blog/a-requirements-challenge
    http://www.scrumcrazy.com/User+Story+Basics+-+What+is+a+User+Story%3F
    http://www.jamesshore.com/Agile-Book/the_planning_game.html
    http://www.cs.toronto.edu/~sme/CSC340F/slides/19-prioritizing.pdf
    http://www.scrumalliance.org/articles/367-its-ordered--not-prioritized
    http://en.wikipedia.org/wiki/Agile_Modeling http://en.wikipedia.org/wiki/Agile_software_development
    http://www.jamesshore.com/Agile-Book/simple_design.html
    http://www.jamesshore.com/Agile-Book/incremental_design.html

www.infosys.com