Use Case Estimation

1. before estimating project size, it need to configure the technical and environmental factors
2. For both TCF (technical complexity factor) and ECF (environment complexity factor), an editable table contains a list of factors influencing project productivity
3. For both TCF (technical complexity factor) and ECF (environment complexity factor), an editable table contains a list of factors influencing project productivity
4. As a suggested gage, a value of '0' indicates no influence, a '3' indicates average influence, and a '5' indicates strong influence.
5. UML use cases help to describe the proposed functionality, which help to assign a rating to each use case.
   a. Easy (5 points): The use case is considered a simple piece of work, uses a simple user interface and touches only a single database entity; its success scenario has less than 3 steps; its implementation involves less than 5 classes
   b. Medium (10 points): The use case is more difficult, involves more interface design and touches 2 or more database entities; its success scenario has between 4 to 7 steps; its implementation involves between 5 to 10 classes
   c. Complex (15 points): The use case is very difficult, involves a complex user interface or processing and touches 3 or more database entities; its success scenario has over seven steps; its implementation involves more than 10 classes
6. The equation is composed of four variables:
   Technical Complexity Factor (TCF).
   Environment Complexity Factor (ECF).
   Unadjusted Use Case Points (UUCP).
   Productivity Factor (PF).

Technical Factor

• **Description**	• **Weight**	**Perceived Complexity**	**Calculated Factor**
T1Distributed System	2	5	10
T2Performance	1	4	4
T3End User Efficiency	1	2	2

Environmental Complexity Factors

Environmental Complexity estimates the impact on productivity that various environmental factors have on an application. Each environmental factor is evaluated and weighted according to its perceived impact and assigned a value between 0 and 5. A rating of 0 means the environmental factor is irrelevant for this project; 3 is average; 5

Environmental Factor	**Weight**
Familiarity with UML	1.5
Stable Requirements	2
Part-time workers	1

Unadjusted Use Case Points (UUCP)

- Unadjusted Use Case Points are computed based on two computations:
- The *Unadjusted Use Case Weight* (UUCW) based on the total number of activities (or steps) contained in all the use case Scenarios.
- The *Unadjusted Actor Weight* (UAW) based on the combined complexity of all the use cases Actors.
- There is an option to include actors in the estimation calculation; by default, only use cases are considered. If project actors are also included, make sure their complexity has been assigned by some method; rough guidelines to this assignment are supplied below:
- Easy: The actor represents another system with a defined API
- Medium: The actor represents another system interacting through a protocol, like TCP/IP
- Complex: The actor is a person interacting via an interface**.**

Computer-Aided Software Engineering [Case**]**

1. CASE stands for Computer Aided Software Engineering which is software that supports one or more software engineering activities within a software development process, and is gradually becoming popular for the development of software as they are improving in the capabilities and **functionality**and are proving to be beneficial for the development of quality software.
2. In other words CASE tools and environments is based on expectations about increasing productivity, improving product quality, facilitating maintenance, and making software engineers' task less odious
3. CASE tools improve documentation quality, improve analysis, and results in systems which help easier to test and maintain.
4. The term Computer-Aided Software Engineering (CASE) encompasses many different products with different functionalities.

5. CASE tools start by considering whether the tool is upper CASE, lower CASE, or integrated CASE
    a. An upper CASE tool (front end CASE) provides support for the early stages in the systems development life cycle such as requirements analysis and design
    b. A lower CASE tool (back end CASE) provides support for the later stages in the life cycle such as code generation and testing.
    c. Integrated CASE tools support both the early and later stages.
6. Further classifications usually list which functionalities are supported by the tool, such as data flow diagrams, entity relationships data models, etc. provides a different type of model of CASE functionality which helps organize CASE tools.
7. CASE Tools offer an excellent array of features that support the development and business community through its Automated Diagram Support feature. The various popular features that aid the development community are listed below:
    - Checks for syntactic correctness
    - Data dictionary support
    - Checks for consistency and completeness
    - Navigation to linked diagrams
    - Layering
    - Requirements traceability
    - Automatic report generation
    - System simulation
    - Performance analysis

8. Case tools can be broadly classed into these broader areas:
    - Requirement Analysis Tool **Requirements analysis** in systems engineering and software engineering, encompasses those tasks that go into

determining the needs or conditions to meet for a new or altered product, taking account of the possibly conflicting requirements of the variousstakeholders, **analyzing, documenting, validating and managing** software or system requirements.

- Structure Analysis Tool are methods for analyzing and converting business requirements into specifications and ultimately,computer programs, hardware configurations and related manual procedures. Context diagram

Data flow diagram

Process specifications

Data dictionary

-
- Software Design Tool **software development tool** is a computer programthat software developers use to create, debug, maintain, or otherwise support other programs and applications.
- Code Generation Tool
- Test Case Generation Tool
- Document Production Tool
- **Reverse Engineering Tool** Reverse engineering is taking apart an object to see how it works in order to duplicate or enhance the object..
-

9. CASE tools play an important role in helping the system developers to perform the task efficiently.

Agile Development – Principles & Practices

- **Agile software development** is a group of software development methods[is a division of software development work into distinct phases (or stages) containing activities with the intent of better planning and management.]in which requirements and solutions evolve through collaboration between self-organizing, cross-functional teams.[
is a group of people with different functional expertise working toward a common goal.[1] It may include people from finance, marketing, operations, and human resources departments]

- The use of the word agile in this context derives from the agile manifesto. A small group of people got together in 2001 to discuss their feelings that the traditional approach to managing software development projects was failing far too often, and there had to be a better way. They came up with the agile manifesto, which describes 4 important values that are as relevant today as they were then. It says, "we value:

- **Individuals and interactions** over Processes and tools
- **Working software** over Comprehensive documentation
- **Customer collaboration** over Contract negotiation
- **Responding to change** over Following a plan
- It promotes adaptive planning, evolutionary development, early delivery, continuous improvement, and encourages rapid and flexible response to change.
- These are characteristics that are common to all agile methods, fundamentally different to a more traditional waterfall approach to software development. They are:
    1. **Active User Involvement Is Imperative**; active user involvement is the first principle of agile development. It's not always possible to have users directly involved in development projects, particularly if the agile development project is to build a product where the real end users will be external customers or consumers.
        - **a.** Requirements are clearly communicated and understood (at a high level) at the outset
        - **b.** Requirements are prioritized appropriately based on the needs of the user and market
        - **c.** Requirements can be clarified on a daily basis with the entire project team, rather than resorting to lengthy documents that aren't read or are misunderstood
        - **d.** Responsibility is shared; the team is responsible together for delivery of the product

e. Timely decisions can be made, about features, priorities, issues, and when the product is ready

2. Agile Development Teams Must Be Empowered:

1. An agile development team must include all the necessary team members to make decisions, and make them on a timely basis.
2. Active user involvement is one of the key principles to enable this, so the user or user representative from the business must be closely involved on a daily basis.
3. The project team must be empowered to make decisions in order to ensure that it is their

4. Responsibility to deliver the product and that they have complete ownership.
5. Any interference with the project team is disruptive and reduces their motivation to deliver.
6. The team must establish and clarify the requirements together, prioritise them together, agree to the tasks required to deliver them together, and estimate the effort involved together

3. **Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.** Have you ever shown your customer software for the first time and received no feedback? In most cases, you receive feedback—sometimes minor, but usually major. The trick is to deliver software early so that you can get feedback early. This early feedback can save you re-work down the road.

4. Business people and developers must work together daily throughout the project. This principle is careful to say business people and not the customer. In most cases, it would be impractical to work with the customer on a daily basis; but generally there are multiple business proxies. These proxies may not know everything about the customer's wants and needs, but they usually know more about the business needs than the developers do. These proxies may be analysts, product managers, or program managers. The key is to maintain constant communication between the developers and the business people to ensure that the project never goes off track—not even for a day

5. **Build projects around motivated individuals.** Give them the environment and support they need, and trust them to get the job done.

Remember, people aren't resources. Software development is different from manufacturing. Software development is more of an art. Project teams need to be motivated and trusted. If you have motivated team members they will find a way to give you their best; and that's what an Agile process needs—everyone's best.

6. **The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.** Instant messaging or the telephone should never replace face-to-face communication. A lot of context is lost in communication over email and instant messaging—not to mention the fact that ambiguity increases with nonverbal communication. Face-to-face communication also lets you run with less formal documentation.

7. **Continuous attention to technical excellence and good design enhances agility.** technical excellence is an essential enabler for a truly Agile software development process. For example, extensible designs and architectures make it much easier to build the product in an evolutionary manner. Automated testing frameworks are needed to ensure that refactoring one part of the system doesn't affect other parts. Continuous integration is essential if you want assurance that your software is working after every change.
8. **Simplicity—the art of maximizing the amount of work not done—is essential.** No code means no bugs. The more code you write, the more bugs your code may have. If something isn't essential to the product, then don't build it. Some developers tend to develop massive underlying frameworks and infrastructures in the system under the assumption that those elements may be needed in the future. The key is simplicity: try not to develop anything that isn't essential to the features you're developing now
9. **The best architectures, requirements, and designs emerge from self-organizing teams.** In traditional software development, analysts write requirements, and architects lay out the architecture of the system. Then the requirements and architectures are communicated to the team in a document. In the Agile world, we encourage teams to self-organize. True self-organization involves giving the whole team the task and asking them, as a team, to complete the task without specifying who should do what—they're left to self-organize. It will naturally occur that architects will lead the discussion when it comes to architecture, but now everyone is free to challenge them and suggest new ideas that may enhance the architecture the architects would have come up with on their own. This form of collaboration also increases the knowledge transfer within the team.
10.

At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly. We believe this is probably the most important principle of agility. The idea of always reflecting on what you're doing and trying to Figure out better ways to do things is the essence of continuous improvement. Without continuous improvement, people and organizations remain at a status quo. If you adopt only one thing that will make your process better, regularly reflect on your process as a team. You need to identify what you're doing well and continue doing it, and you need to identify what you're doing poorly and improve it.

AGILE PRACTISE

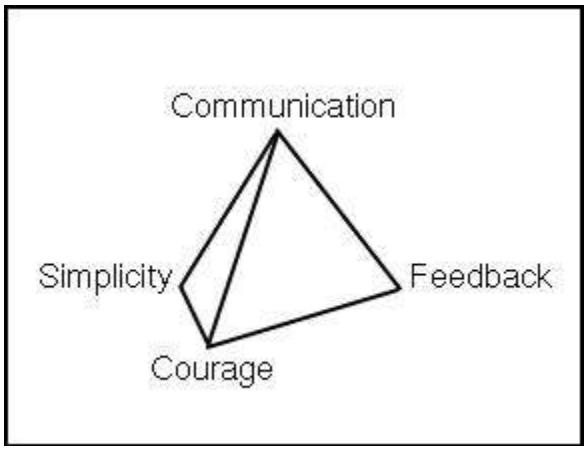
- Activities that are used to manifest or implement the agile principles and values.
- There are numerous agile practices, such as user stories, test-driven development, pair programming, daily stand-up meetings, and so forth.
- Organizations create different Agile practices or tailor existing Agile practices to address specific organizational or team needs. Teams may also need to be creative and come up with new Agile practices to achieve agility while adhering to organizational constraints.
- There are various methodologies that are collectively known as agile, as they promote the values of the agile manifesto and they are consistent with the above principles.
- 1. DSDM is probably the original agile development method. DSDM was around before the term 'agile' was even invented, but is absolutely based on all the principles we've come to know as agile.
- 2. Scrum is also an agile development method, which concentrates particularly on how to manage tasks within a team-based development environment. Scrum is the most popular and widely adopted agile method
- 3. XP (Extreme Programming) is a more radical agile methodology, focusing more on the software engineering process and addressing the analysis, development and test phases with novel approaches that make a substantial difference to the quality of the end product.

Extreme programming- Core values & Practices Frameworks

Extreme Programming (XP) is an agile software-development methodology.

Extreme Programming (XP) is a pragmatic approach to program development that emphasizes business results first and takes an incremental, get-something-started approach to building the product, using continual testing and revision.

CORE VALUES



XP is built on four core values: communication, simplicity, feedback, and courage.

1. Communication

Software is developed as quickly as the communication links in the project allow. The customer communicates her requirements to programmers. The programmers communicate their interpretation of the requirements to the computer. The computer communicates with its users. The users communicate their satisfaction with the software to the customer.

Communication in XP is bidirectional and is based on a system of small feedback loops. The customer asks the users what they want. The programmers explain technical difficulties and ask questions about the requirements. The computer notifies the programmers of program errors and test results.

In an XP project, the communication rules are simple: all channels are open at all times. The customer is free to talk to the programmers. Programmers talk to the customer and users. Unfettered communication mitigates project risk by reducing false expectations. All stakeholders know what they can expect from the rest of the team.

2. Simplicity

XP takes simplicity to the extreme with practical guidelines:

- Do the simplest thing that could possibly work (DTSTTCPW),
- Represent concepts once and only once (OAOO),

- You aren't going to need it (YAGNI), and
- Remove unused function.

Feedback

The more immediate feedback, the more efficiently a system functions.

The long delay in the system makes showering unpleasant and inefficient.

For many customers, this is what software development is like. You request a change, and it is delivered many months later in some big release. Often the change fails to meet your expectations, which means another change request with yet another long delay.

XP is like a well-designed shower. You request a change and out comes software. Adjustments are visible immediately. The customer sees her requirements or corrections implemented within weeks. Programmers integrate their changes every few hours, and receive code reviews and test results every few minutes. Users see new versions every month

The value of immediate, real world feedback should not be underestimated. One of the reasons for the success of the Web is the abundance of structured and immediate feedback from users. Developers see errors in real time, and contract all input and output that causes Web application failures.

XP reduces project risk by taking iterative development to the extreme. The customer's involvement does not end at the planning phase, so requirements errors are reconciled almost immediately. The system's internal quality is maintained by programmers working in pairs who are striving for simplicity. Automated testing gives everybody feedback on how well the system is meeting expectations.

XP uses feedback to integrate towards a solution, rather than trying to get it through a discontinuity

Courage

XP teams (expected to) have the courage to communicate and accept feedback. They have the courage to throw code away (prototypes), and re-factor the architecture of a system. Design, architecture or prototype changes

are no longer a worry for Software development process, do you see the change?

Respect

without which you can never be successful is our last value,
Respect. Everyone gives and feels the respect they deserve as a valued team member. Everyone contributes value even if it's simply
enthusiasm. Developers respect the expertise of the customers and vice versa. Management respects the team’s right to accept responsibility and receive authority

DESIGN PATTERNS

A design pattern is a repeatable solution to a software engineering problem. Unlike most program-specific solutions, design patterns are used in many programs. Design patterns are not considered finished product; rather, they are templates that can be applied to multiple situations and can be improved over time, making a very robust software engineering tool. Because development speed is increased when using a proven prototype, developers using design pattern templates can improve coding efficiency and final product readability.

Successful designs and architectures within design patterns make software development easier. Establishing proven design pattern techniques makes them readily available for future system developers. In system documentation, design patterns are also by improving existing system maintenance to create a better help tool. By providing explicit specifications of class and object interaction and the interaction’s original intent, design patterns help developers correctly work through the application.

Examples of Design Patterns

Design patterns are easiest understood when looking at concrete examples. For beginners the following ten patterns may suffice. However, you should make it a habit to learn about some of the other patterns mentioned below. The more patterns you know the better.

1. Factory Method

The Factory pattern creates an object from a set of similar classes, based on some parameter, usually a string.

2. **Abstract Factory**[edit]

Where the Factory pattern only affects one class, the Abstract Factory pattern affects a whole bunch of classes.

3. **Singleton**[edit]

This is one of the most dangerous design patterns, when in doubt don't use it. Its main purpose is to guarantee that only one instance of a particular object exists. Possible applications are a printer manager or a database connection manager. It is useful when access to a limited resource needs to be controlled.

4. **Iterator**[edit]

Nowadays, the Iterator pattern is trivial: it allows you to go through a list of objects, starting at the beginning, iterating through the list one element after the other, until reaching the end

5. Template Method[edit]

Also the Template Method pattern is rather simple: as soon as you define an abstract class, that forces its subclasses to implement some method,

6. Command[edit]

To understand the idea behind the Command pattern consider the following restaurant example: A customer goes to a restaurant and orders some food. The waiter takes the order (command, in this case) and hands it to the cook in the kitchen. In the kitchen the command is executed, and depending on the command different food or drink is being prepared

7. The Observer pattern is one of the most popular patterns, and it has many variants. Assume you have a table in a spreadsheet. That data can be displayed in table form, but also in form of some graph or histogram. If the underlying data changes, not only the table view has to change, but you also expect the histogram to change. To communicate these changes you can use the Observer pattern: the underlying data is the *observable* and the table view as well as the histogram view are *observers* that observe the observable.

Composite[edit]

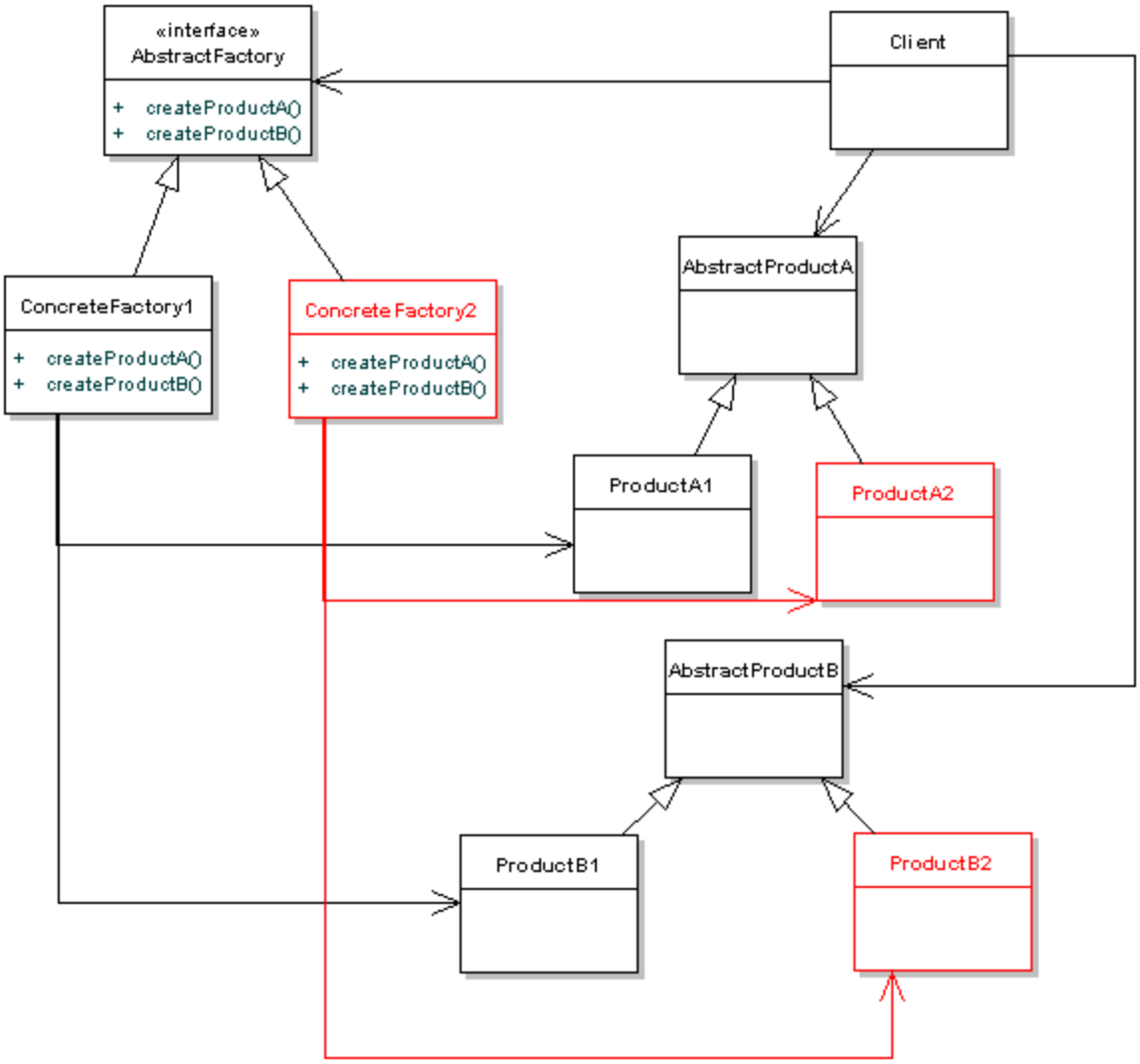
The Composite pattern is very wide spread. Basically, it is a list that may contain objects, but also lists. A typical example is a file system, which may consist of directories and files. Here directories may contain files, but also may contain other directories

State[edit]

In the State pattern, an internal state of the object influences its behavior. Assume you have some drawing program in which you want to be able to draw straight lines and dotted lines. Instead of

creating different classes for lines, you have one Line class that has an internal state called 'dotted' or 'straight' and depending on this internal state either dotted lines or straight lines are drawn

EXAMPLE



1. The **AbstractFactory** defines the interface that all of the concrete factories will need to implement in order to product **Product**s
2. **ConcreteFactoryA** and **ConcreteFactoryB** have both implemented this interface here, creating two seperate families of product
3. Meanwhile,**AbstractProductA** and **AbstractProduct B** are interfaces for the different types of product.

4. The **Client** deals with **AbstractFactory**, **AbstractProductA** and **AbstractProductB**.
5. It doesn't know anything about the implementations. The actual implementation of **AbstractFactory** that the **Client** uses is determined at runtime.
6. As you can see, one of the main benefits of this pattern is that the client is totally decoupled from the concrete products
7. new product families can be easily added into the system, by just adding in a new type of **ConcreteFactory** that implements **AbstractFactory**, and creating the specific Product implementations.
8.