**SDLC Models**

Table of Contents

[Problems with Software Development 3](#_Toc83392935)

[Constraints of Software Development Projects 4](#_Toc83392936)

[Popular SDLC Models 5](#_Toc83392937)

[Waterfall Model 6](#_Toc83392938)

[V-Shaped Model 8](#_Toc83392939)

[Iterative Incremental Model 10](#_Toc83392940)

[Spiral Model 12](#_Toc83392941)

[Spiral Areas 12](#_Toc83392942)

[Steps 13](#_Toc83392943)

[Prototyping Model 16](#_Toc83392944)

[Throwaway or Rapid Prototyping 16](#_Toc83392945)

[Evolutionary Prototyping 16](#_Toc83392946)

[Incremental Prototyping 17](#_Toc83392947)

[Extreme Prototyping 17](#_Toc83392948)

[Agile Development 20](#_Toc83392949)

[Agile Methodologies 21](#_Toc83392950)

[Agile Manifesto 22](#_Toc83392951)

[Scrum 25](#_Toc83392952)

[XP 26](#_Toc83392953)

[Lean 29](#_Toc83392954)

[Kanban 30](#_Toc83392955)

[Feature-Driven Development 31](#_Toc83392956)

[Dynamic System Development Method 32](#_Toc83392957)

[Crystal 33](#_Toc83392958)

[Selecting the Right SDLC 34](#_Toc83392959)

## Problems with Software Development

* **Requirements are complex**. The client does not know the functional requirements in advance.
* **Requirements change**. This happens so frequently that many companies have a process laid out to deal with requirement changes.
* **Frequent changes** are difficult to manage.
* There is **more than one software system**. New systems must be backwards compatible with existing systems (legacy systems).

## Constraints of Software Development Projects

* Computer Resources
* Time
* Money
* Staff – This includes programmers, designers, managers, etc.

## Popular SDLC Models

* Waterfall Model
* V-Shaped Model
* Iterative Incremental Model
* Spiral Model
* Prototyping Model
* Agile development

## Waterfall Model

The Waterfall Model was originally described in 1970 as an example of a **flawed**, **non-working** model. So obviously, it is the most widely used SDLC model in the world. 😀

The Waterfall Model has **5 stages**.

1. Requirements
2. Design
3. Implementation
4. Verification
5. Maintenance

The reason the model is called a ‘waterfall’ model is because it is **not possible to go back** to any previous phase. It is a **sequential** approach, where a single phase must be finished completely before moving forwards. In fact, the process to go back to the previous phase is not even defined.

The waterfall model can provide structure to inexperienced staff. We should use the Waterfall Model when:

* The **requirements** are **clear** and **frozen**.
* **Technology is well understood** by the team.
* The project cannot be delivered in an **iterative manner**.
* **Documentation** is essential.
* Professional **project management skills** are available.
* The project **cost** is defined. This could not have been done if requirements were changing.

Advantages:

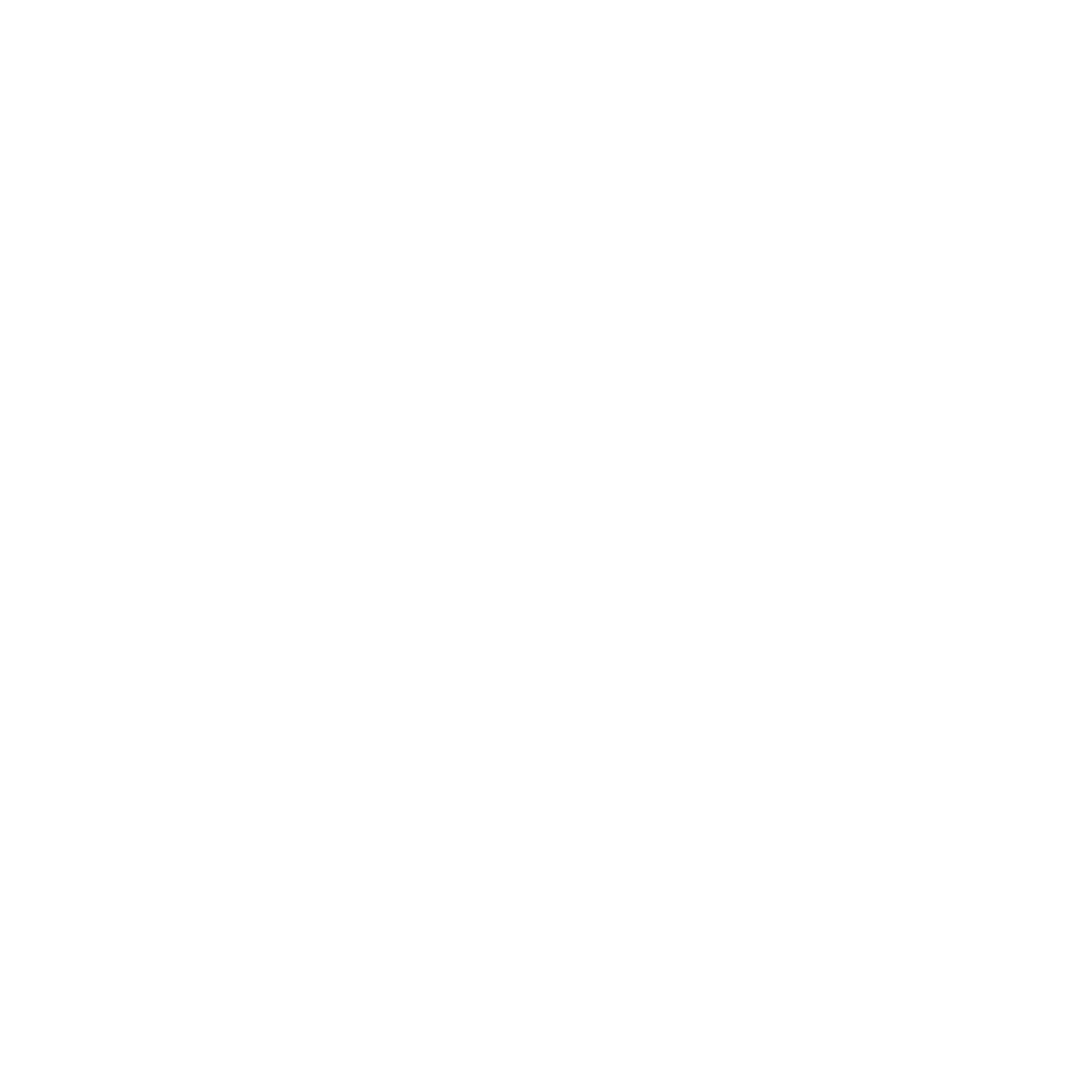
* It is very **easy to explain** to the business users and explain the output of each phase.
* It has a **structured approach**.
* The stages and activities are **well defined**.
* It is easier for project managers to **plan**, **schedule** the project, **utilize** **resources**, and **define milestones**.
* There is validation and verification at each phase to ensure **early detection of errors** or misunderstandings at the same phase.
* Each phase has **specific deliverables**.

Disadvantages

* It takes the **full lifecycle** to deliver a workable solution to the customer.
* It is very difficult to **go back** to any phase after it has finished.
* It assumes that the **requirements** of a system can be **frozen** without any changes or enhancements.
* Even a little **flexibility** and adjustment is **difficult** and expensive.
* It requires **more time** for the detailed plan upfront of the project, as the requirements are clear and frozen and it should be visible to have the detailed plan delivered to the customer.
* It **delays the testing phase** which can discover a lot of issues in requirements, design, and implementation as well.

## V-Shaped Model

The **V-Shaped Model** is an extension of the Waterfall Model. **All the phases** that exist in the Waterfall Model **also exist** in the V-Shaped Model. The major difference is that we are allowed to **plan** the tests for the phase we just completed.



For example, directly after the **Requirement Analysis** phase, we can plan the **Acceptance Tests**. These are tests that check whether the **requirements** set by the client are **met** or not. The actual test occurs at the end of the model, but we can plan it now.

We should use the V-Shaped Model when:

* The **requirements** are well defined, clearly documented and **fixed**.
* The **product definition** is **stable**.
* The **technology** being used is **not dynamic** and is **well understood** by the project team.
* There are no ambiguous or **no undefined requirements**.
* The **project is short** in nature.

Advantages:

* It is **simple** and easy to use.
* Each phase has **specific deliverables**.
* It has a **higher chance of success** over the waterfall model due to the development of test plans early on during the life cycle.
* It **works well** for where **requirements** are easily **understood**.
* There is **verification and validation** of the product in the **early stages** of product development.

Disadvantages:

* It is very **inflexible**, like the waterfall model.
* The **adjustment** scope is **difficult** and **expensive**.
* The software is developed during the implementation phase, so **no early prototypes** of the software are produced.
* The model does not provide a clear path for **problems found** during testing phases.
* **Costly** and requires **more time**, in addition to a **detailed plan**.

## Iterative Incremental Model

The **Iterative Incremental Model** was developed to **overcome the weaknesses** of the Waterfall Model. It starts with an **Initial Planning** stage and ends with a **Deployment** stage, but everything in between happens in a **cyclic** manner.

The basic idea is to develop the system through **repeated iterations** (iterative) and in **smaller portions** (incremental) at a time. This allows developers to take **advantage** of what they learnt during the development of **earlier parts**. The cycles may consist of **mini-Waterfall Models** or mini-V-Shaped Models.

We should use the Iterative Incremental Model when:

* Most of the requirements are **known up-front** but are expected to **evolve**.
* The requirements are **prioritized**.
* There is a need to get the basic functionality **delivered fast**.
* A project has **lengthy development schedules**.
* A project has **new technology** or the **domain is new** to the team.

Advantages:

* **Prioritized requirements** can be developed first.
* The **initial product delivery is faster**.
* Customers gets important functionality **early**.
* It **lowers initial delivery cost**.
* Each release is a product increment, so that the customer will have a **working product** at hand all the time.
* The customer can provide **feedback** to each product increment, thus **avoiding surprises** at the end of development.
* Requirements **changes** can be easily **accommodated**.

Disadvantages:

* It requires **effective planning** of iterations.
* It requires **efficient design** to ensure inclusion of the required functionality and provision for changes later.
* It requires **early definition** of a complete and fully functional system to allow the definition of increments.
* **Well-defined module interfaces** are required, as some are developed long before others are developed.

## Spiral Model

The **Spiral Model** is a combination of the Waterfall Model and Iterative Model. Each phase in the spiral model begins with a **design goal** and ends with the client **reviewing** the progress.

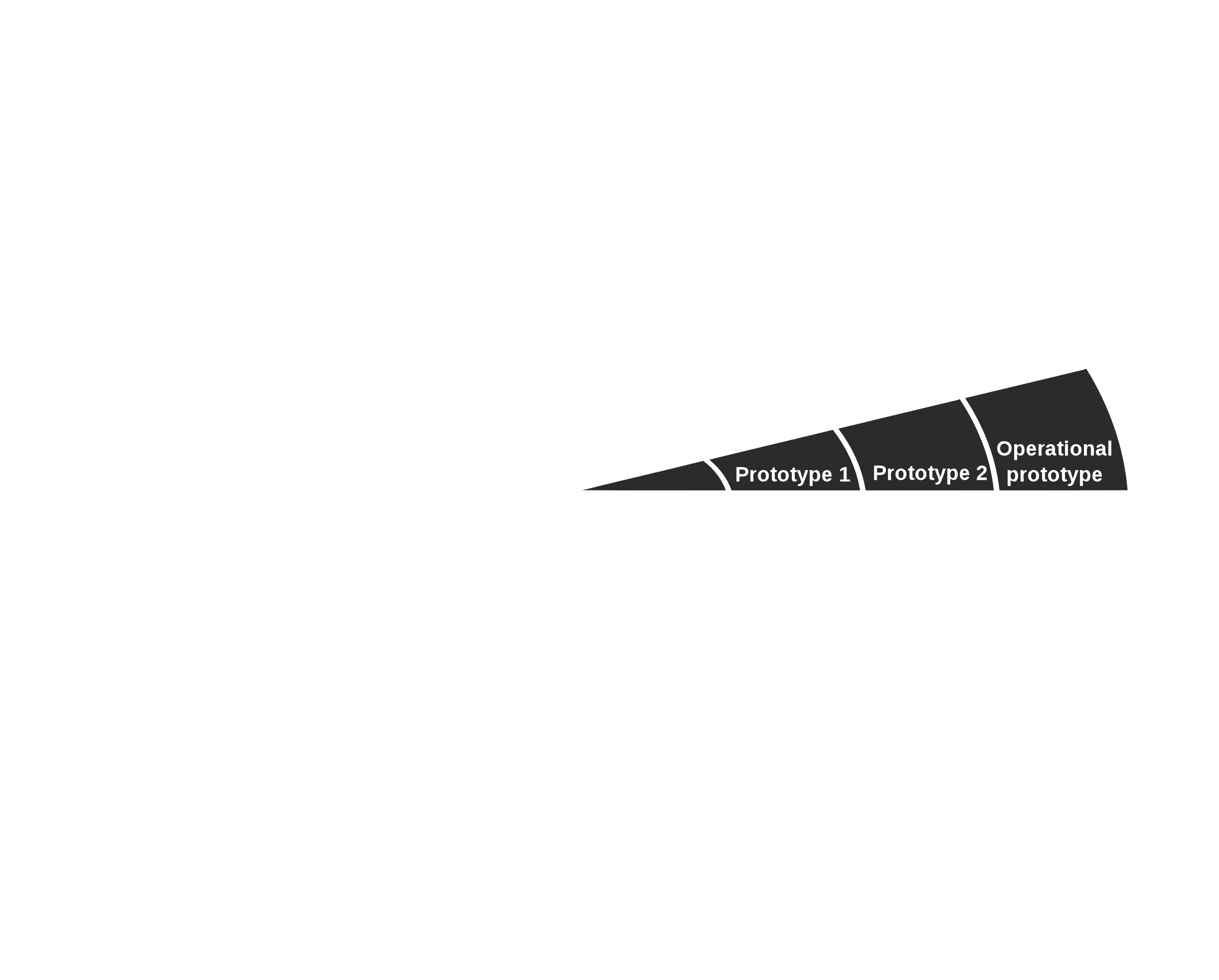
The development team starts with a **small set of requirements** and goes through each development phase for those set of requirements. The software engineering team **adds functionality** for the additional requirements in ever-increasing spirals until the application is ready for the production phase.

### Spiral Areas

There are mainly **four** areas in the Spiral Model.

1. **Planning** – This includes estimating the **cost**, **schedule** and **resources** for the iteration. It also involves understanding the **system requirements** for continuous communication between the system analyst and the customer.
2. **Risk Analysis** – Potential risks must be **identified** and **risk mitigation strategies** must be planned and finalized.
3. **Engineering** – This includes **testing**, **coding** and **deploying** software at the customer site.
4. **Evaluation** – The customer **evaluates** the software. Additionally, risks like **schedule slippages** and **cost overruns** must be monitored.

Diagrammatically, the phases are traversed like this:



Once the development stage has started, it is called the **point of no return**. We cannot go backwards for that iteration anymore.

### Steps

1. **Define requirements** with user involvement and analysis of existing system
2. Initial new **system design**
3. Construct and evaluate an **initial prototype**, a rough (skeletal) system attribute framework.
4. Construct a further **(refined) prototype**, basing it on evaluation of initial prototype, defining its scope, planning its development and implementing it
5. Overall (system-wide) **risk assessment**
6. **Prototype assessment** (as per step 4) and possible development of further prototypes
7. **Repeat steps 1-5** until refined prototype meets user expectations
8. **Construct the system** (based on final refined prototype)
9. Test and maintain the system

We should use the Spiral Model when:

* The project is **large**
* **Releases** are required to be **frequent**
* Creation of a **prototype** is applicable
* **Risk and costs evaluation** is important
* The project is **medium to high-risk**
* Requirements are **unclear** and complex
* **Changes** may be required at any time

Advantages:

* Additional functionality or **changes** can be done at a **later stage**
* **Cost estimation becomes easy** as the prototype building is done in small fragments
* Continuous or repeated development helps in **risk management**
* **Development is fast** and features are added in a systematic way
* There is always space for **customer feedback**

Disadvantages:

* There is a risk of **not meeting the schedule or budget**
* It works best for **large projects only** and also demands **risk assessment expertise**
* For its smooth operation, **spiral model protocol** needs to be **followed strictly**
* There is **more documentation** as it has intermediate phases
* It is **not advisable for smaller project**, since it might **cost** a lot

## Prototyping Model

**Prototyping** is the process of quickly putting together a working model (a prototype) in order to test various aspects of a design, illustrate ideas or features and gather early user feedback.

There are **four** major software prototyping types:

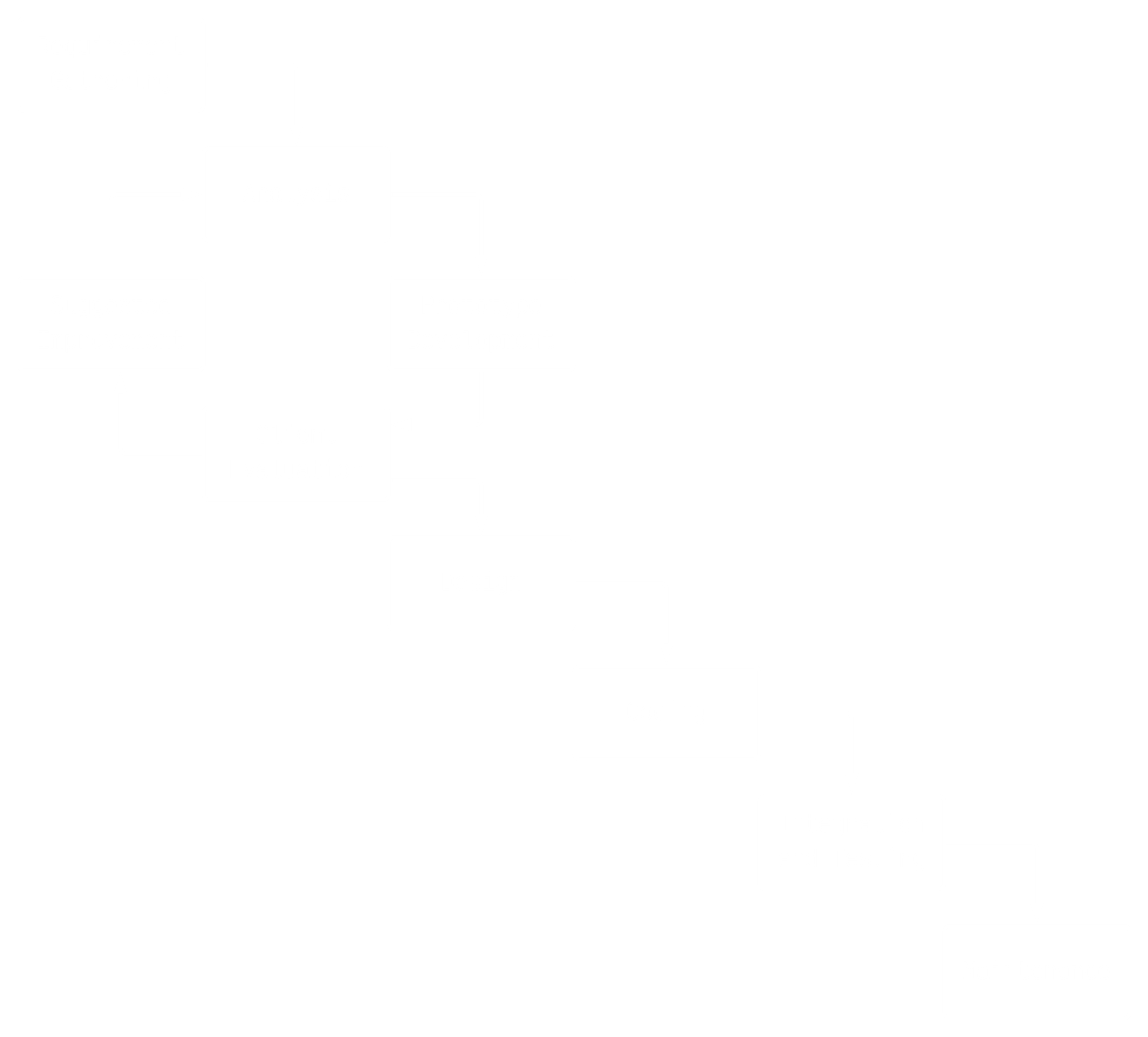
1. Throwaway or Rapid Prototyping
2. Evolutionary Prototyping
3. Incremental Prototyping
4. Extreme Prototyping

### Throwaway or Rapid Prototyping

In **Rapid Prototyping**, we use very little effort to build a prototype that has just the bare **minimum features** initially gathered from **minimal requirement analysis**. Once the actual requirements are understood, this prototype is **discarded** and the actual system can be developed with a far clearer understanding of user requirements.

### Evolutionary Prototyping

**Evolutionary Prototypes** are actual, **functional** prototypes that just have minimal functionality to start with. Once **well-understood requirements** are included and more are added when they are understood.



### Incremental Prototyping

In **Incremental Prototyping**, multiple functional prototypes of the **different subsystems** are built and later **integrated** to create the complete system.

### Extreme Prototyping

**Extreme Prototyping** is for **Web Development**. It consists of three phases.

1. A **basic** prototype with the existing pages is presented in **HTML** format.
2. **Data Processing** is simulated using a **prototype services layer**.
3. Services are **implemented** and **integrated** in the final prototype.

Prototyping is used:

* When the **requirements** are **unclear**
* If requirements are **changing quickly**
* To develop user interfaces, high technology software-intensive systems, and systems with **complex algorithms and interfaces**
* To determine the **technical feasibility** of the product

Advantages:

* **Errors** can be detected in the **initial stage** of the development process
* **Missing functionality** can be **identified**, which reduces the risk of failure
* **Customer satisfaction** exists because the customer can feel the product at a very early stage.
* As customers are involved from the early stage, there will be hardly any chance of **software rejection**
* **Quicker user feedback** helps achieve better solutions
* It is a **straightforward** model, so it is easy to understand.
* No need for **specialized experts** to build the model
* The prototype serves as a **basis** for deriving a system specification
* The prototype helps to gain a **better understanding** of the customer's needs
* Prototypes may offer **early training** for future users of the software system

Disadvantages:

* Lots of **variation in requirements** each time the prototype is evaluated
* **Poor documentation** due to continuously changing customer requirements
* Difficult for the developers to **accommodate all the changes** demanded
* There is uncertainty in determining the **number of iterations** that would be required before the prototype is finally accepted by the customer
* After seeing an early prototype, the customers sometimes **demand the actual product** to be delivered quickly
* Developers hurrying to build prototypes may create **sub-optimal solutions**
* The customer might **lose interest** in the product if he/she is **not satisfied** with the **initial prototype**

Based on all this, the final question is, do we need prototyping? The answer tends to be **yes**, mainly due to two factors:

1. They reduce the **cost** and **time-to-market** of a system.
2. For companies building **critical systems**, prototyping would help them perform **formal verification** when required. These methodologies provide a high level of **reliability** in the system design and implementation.

## Agile Development

Some of the models we have seen before were able to adjust to changes through the use of iterations. However, this is not fast enough. Those models simply took feedback and implemented features in the next iteration. Nowadays, it is expected that priorities will be shifted and changes will be implemented immediately. This is where **Agile Development** comes in.

Agility is:

* The ability to create and **respond to change** in order to profit
* The ability to **quickly reprioritize** use of resources when requirements, technology, and knowledge shift
* The use of evolutionary, incremental, and iterative delivery to converge on an **optimal customer solution**
* Maximizing **business value** with right sized, just enough, and just-in-time processes and documentation

**Agile** is a set of practices, values and principles for software development. The core ideas of Agile Development are:

* **Adaptability**, meaning the teams and the process should be able to adjust to rapid changes.
* **Incremental and Iterative** workflow, meaning products are produced in stages, a growing set of complete and working software.
* Being ‘**People-Oriented**’, meaning the people involved in the project are the priority.

In the previous models, our goal was to **provide** **a product**. We achieved this in several different ways. In agile however, we take a vision of building a product and a good team of people and aim to create an **improved team**. Essentially, the team needs to be able to be better at agile development at the end of the project.

One example could be a situation where a member from a different team needs to be contacted urgently. A traditional process would involve emails and phone calls, all of which could potentially take more time than we have at hand. Agile skips over these formalities and encourages team members to simply go over and talk immediately, thus saving a huge amount of time.

At the end of each iteration, called a **sprint** in agile, we may have a potentially releasable product. Even in the middle of a sprint, we can accommodate changes. We do this by reprioritizing our tasks.

### Agile Methodologies

There are actually several Agile Development Methodologies:

* Scrum
* Kanban
* XP
* FDD
* DSDM
* Lean
* Crystal

### Agile Manifesto

The **Agile Manifesto** states:

* Individuals and interactions over processes and tools
* Working software over comprehensive documentation
* Customer collaboration over contract negotiation
* Responding to change over following a plan

Although this may make it sound like there is no documentation, process or plan, that is entirely untrue. Those things still exist, but are simply given less priority.

The complete Agile Manifesto is given below:

1. The highest priority is to satisfy the customer through early and continuous delivery of software.
2. Welcome changing requirements, even late in development.
3. Deliver working software frequently, in a few of weeks to a few of months.
4. Business people and developers must work together daily.
5. Build projects around motivated individuals. Provide the environment and support they need.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development at a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity, the art of maximizing the amount of work not done, is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.

We should use agile when:

* **New changes** are need to be implemented and the project needs to be delivered in a **short amount of time**.
* The project involves **iterative**, or cyclical, processes in which **incremental** results will add value to the project.
* There is some **budget flexibility**
* Trying to build something **innovative** that does not exist
* There is high **product owner involvement**.
* The organization does not have **strict processes** to follow and there is the luxury of being able to work flexibly

Advantages:

* **Customer satisfaction** by rapid, continuous delivery of useful software.
* **People and interactions** are emphasized rather than process and tools.
* Customers, developers and testers constantly **interact** with each other.
* **Working software** is delivered frequently (weeks rather than months).
* **Face-to-face conversation** is the best form of communication.
* Close, daily **cooperation** between business people and developers.
* Continuous attention to **technical excellence** and **good design**.
* Regular **adaptation** to changing circumstances.
* Even **late changes** in requirements are welcomed

Disadvantages

* In case of some software deliverables, especially the large ones, it is **difficult to assess the effort required** at the beginning of the software development life cycle.
* There is **lack of emphasis** on necessary **designing and documentation**.
* The project can easily get taken **off track** if the customer representative is not clear what final outcome they want.
* Only **senior programmers** are capable of taking the kind of decisions required during the development process. Hence, it has no place for newbie programmers, unless combined with **experienced resources**.

## Scrum

Scrum is the most commonly used Agile methodology. It allows us to focus on delivering the **highest business value** in the **shortest time** and rapidly and repeatedly **inspect actual working software**, every two weeks to one-month in fact.

The business sets the **priorities**. Teams self-organize to determine the best way to deliver the highest priority features. Every two-weeks to a month, anyone can see **real working software** and decide to release it as is or continue to enhance it for another sprint.

## XP

**Extreme Programming**, more commonly called XP, is a disciplined approach to delivering high-quality software quickly and continuously. It is intended to improve **software quality** and **responsiveness** in the face of changing customer requirements.

The original XP method is based on **four** simple values:

1. Simplicity
2. Communication
3. Feedback
4. Courage

It also has **twelve supporting practices**:

1. **Planning Game** – This refers to planning for the **upcoming iteration** and **user stories** provided by the customer. Technical persons determine schedules, estimates, costs, etc. It is a result of **collaboration** between the customer and the developers.
2. **Small Releases** – The releases should be small in terms of **functionality**. Less functionality means releases happen more frequently. This practice supports the Planning Game.
3. **Metaphor** – Metaphors are used to explain the **architecture** of the system. This makes it quick and easy and encourages a **common set of terms** for the system.
4. **Simple Design** – We should do **as little as needed**. This makes it easier to **understand** what is happening and helps keep programmers **on-track**.
5. **Testing** – **Unit testing** is used and a **Test-first** design is adopted, meaning testing is planned out first. All **regression testing** is automated.
6. **Refactoring** – This is the process of **changing how** the system does something but **not what** is done. It improves the **quality** of the system in some way and increases **developer knowledge** of the system
7. **Pair Programming** – **Two developers** work on one monitor, with one keyboard. One actually codes while the other thinks and they switch roles as needed. Two people are more likely to answer the following questions:
8. Is this whole approach going to work?
9. What are some test cases that may not work yet?
10. Is there a way to simplify this?
11. **Collective Ownership** – This is the idea that all developers own all of the code and thus understand it. It enables the refactoring practice and helps **mitigate the loss** of a team member leaving. It promotes developers to take **responsibility** for the system as a whole rather than parts of the system.
12. **Continuous Integration** - New features and changes are worked into the system **immediately**. Code is not worked on without being integrated for more than a day.
13. **40-Hour Workweek** - The work week should be limited to **40 hours**. Regular overtime is a symptom of a problem and not a long-term solution. Most developers **lose effectiveness** past 40-Hours. Thus, value is placed on the developers’ **well-being**, and management is forced to find **real solutions**.
14. **On-site Customer** – This allows for quick and continuous **feedback** to the development team. The customer can give **quick and knowledgeable answers** to real development questions and make sure that what is developed is what is needed. **Functionality** is prioritized correctly
15. **Coding Standards** - All code should look the **same**. It should not be possible to determine who coded what based on the code itself. This reduces the amount of **time** developers spend reformatting other peoples’ code.

## Lean

Lean was originally developed by Toyota to improve their manufacturing process by eliminating waste, improving processes and boosting innovation. It was later adopted by other fields as well. Software development is a natural application of Lean methodology because, much like manufacturing, it generally follows a defined process, has some **defined conditions of acceptance,** and results in the delivery of **tangible value**.

The key concepts that guide all practice of Lean methodology, which are called the **Pillars of Lean**, are:

* Continuous improvement
* Respect for people
* Lightweight Leadership

## Kanban

Kanban is a highly visual workflow management method. It is based on 3 basic principles:

1. **Visualize** what you’ll do today (workflow) - Seeing all the items within the context of each other can be very informative.
2. **Limit** the amount of work in progress (WIP) - This helps balance the flow-based approach so teams don’t start and commit to too much work at once.
3. **Enhance** flow - When something is finished, the next highest priority item from the backlog is pulled into play.

## Feature-Driven Development

Feature-Driven Development (**FDD**) is an iterative and incremental software development process and is an **agile method** for developing software. It allows teams to **update** the project regularly and **identify errors** quickly. Its practices are driven from a **client-valued** functionality (**feature**) perspective. The main purpose is to deliver tangible, working software repeatedly in a timely manner.

It consists of **five basic activities**:

1. **Development** of an overall model
2. **Building** of a feature list
3. **Planning** by feature, meaning high-priority features go first
4. **Designing** by feature
5. **Building** by feature

## Dynamic System Development Method

The Dynamic System Development Method (DSDM) is a framework that is made up of **eight principles** and **several best practice techniques**. It prioritizes **schedule** and **quality** over functionality, which fixes cost, quality and time at the start and uses the **MoSCoW** method of prioritization, which breaks a project down into four different types of requirements:

1. Must have (Mo)
2. Should have (S)
3. Could have (Co)
4. Won’t have (W)

The **eight principles** of DSDM direct the team in the attitude they must take and the mindset they must adopt to deliver consistently.

1. Focus on the business need
2. Deliver on time
3. Collaborate
4. Never compromise quality
5. Build incrementally from firm foundations
6. Develop iteratively
7. Communicate continuously and clearly
8. Demonstrate control

## Crystal

Crystal methods are focused on:

1. People
2. Interaction
3. Community
4. Skills
5. Talents
6. Communications

## Selecting the Right SDLC

Selecting the right SDLC model involves several steps:

1. **Learn SDLC** – We need to have knowledge about the different SDLC models available to us.
2. **Stakeholder Needs** – We need to consider the business domain, stakeholder concerns and requirements and business priorities.
3. **Define Criteria** – We need to check a whole host of things about the team and the project. Is the SDLC model suitable for the team size and their skills? Is the technology appropriate? Is the size and complexity of the software appropriate?
4. **Decide** – Taking all of the above into consideration, we need to pick an SDLC model. This could involve assigning weights and scores to each criterion.
5. **Optimize** – Every SDLC model can be improved upon. We need to optimize the model based on our needs. We could even end up creating our own SDLC model.

Doing all this work is hard, but it has its benefits. Picking the right SDLC model can:

* Increase **development speed** and decrease time to market
* Improve **product quality**
* Increase **project visibility**, making the project easier to understand for everyone involved
* Decrease **administrative overhead**
* Make **risks** obvious
* Improve **customer relations**