# **SOFTWARE DEVELOPMENT LIFE CYCLE Document**

## Introduction:

Via Sendify, the project team essentially wishes to create a custom notification system as one of the features of an end product. The aim is to abstract out the hassle of building infrastructure for such a system, and provide developers with easy APIs and SDKs which they can quickly integrate into the existing application and create a notification system with only a few lines of code.

The team has a basic idea of the quintessential functional and technical requirements of the project, but acknowledges that certain features may need to be added/modified/removed as the software progresses through the various development stages. We believe that such an issue could be addressed by adhering to the following operating guidelines:

The developers wouldn’t implement the poorly understood features. The partial system would be sent to customer sites and as users work with the system, they would be able to detect opportunities for new features and give requests for these features to the developers. The developers could then take these enhancement requests along with their own and change the software-requirements specification, update the design, recode and retest.

## Accepted Software Development Life Cycle (SDLC) Model:

Several SDLC models were considered for adoption by the project team. However, after a careful analysis of the positives and negatives associated with each and every model, we closed down on a hybrid of ‘extreme programming’ and ‘evolutionary software prototyping’ as the most suitable choice for our project:

As a type of agile software development, extreme programming advocates frequent ‘releases’ (alternatively ‘software builds’) in short development cycles, which is intended to improve productivity and introduce checkpoints at which new customer requirements can be adopted. Other features of extreme programming include:

* doing extensive code review.
* unit testing of all code.
* avoiding programming of features until they are actually needed.
* expecting changes in the customer's requirements as time passes and the problem is better understood.
* frequent communication with the customer and among programmers.

However, by definition, the release/build after every development cycle may only yield modified, working modules and not necessarily a usable application prototype. This poses a severe threat to the success of the project since Sendify doesn’t attempt to tweak around the features of an existing software, but develop an entirely new one altogether. To address such an issue, the project team consensually decided that every development cycle should yield a ‘working prototype’ to allow users of the software to evaluate developers' proposals for the design of the eventual product by actually trying them out, rather than having to interpret and evaluate the design based on descriptions. Such prototypes can also be used by end users to describe and prove requirements that developers have not yet considered.

Each working prototype would typically be required to simulate only a few aspects of the application and may be completely different from the final product.

This particular model has the following benefits to offer:

* The software designer and implementer can get valuable feedback from the users early in the project.
* The client and the contractor can compare if the software made matches the software specification, according to which the software program is built.
* It also allows the software engineer some insight into the accuracy of initial project estimates and whether the deadlines and milestones proposed can be successfully met or not.

The reason why the team wishes to choose **‘evolutionary prototyping’** is that it admits that it may not understand all the requirements at first and therefore should build only those that are well understood. Using evolutionary prototyping, the team can build a very robust prototype in a structured manner and constantly refine it. The evolutionary prototype, when built, will form the heart of the new system, and improvements will be made as and when further requirements are specified and understood. This technique would allow the development team to add features to the initial prototypes, or make changes that couldn't be conceived during the requirements and design phase. During the early stages, the prototypes may not have all the features the users have planned, but they may be used on an interim basis until the final system is delivered.

## 

## Reasons for rejecting other models:

Several other SDLC models were considered by the development team and rejected for one reason or the other. These included:

* Waterfall model
* Iterative model
* Spiral model
* V-model
* Big Bang model
* RAD model

Since there is no specific process followed in the Big Bang model, and the output may or may not be as per customer requirement, it was rejected by the project team in the first go itself.

Other negatives associated with the model are as follows:

* Very High risk and uncertainty.
* Not a good model for complex and object-oriented projects.
* Poor model for long and ongoing projects.
* Can turn out to be very expensive if requirements are misunderstood.

The Waterfall and V-models would have rendered an overall structure to the project and allowed for departmentalization and control, but wouldn’t have provided any room for reflection or revision: Once the application would have reached the testing phase, it would have been very difficult to go back and change something that wasn’t well-documented or thought upon in the concept stage.

The following are the other negatives associated with the model:

* No working software is produced until late during the life cycle.
* Not a good model for complex and object-oriented projects.
* Poor model for long and ongoing projects.
* Not suitable for the projects where requirements are at a moderate to high risk of changing. So risk and uncertainty is high with this process model.

Since the team has gone for an agile model, which is a combination of iterative and incremental process models, and has extra features such as focus on process adaptability and customer satisfaction by rapid delivery of working software product, there wasn’t much sense in limiting the development to only the iterative model.

The spiral model combines the idea of iterative development with the systematic and controlled aspects of the waterfall/v-model, thereby addressing many of the risks identified by the development team, since:

* Changing requirements can be accommodated.
* Requirements can be captured more accurately.
* Users see the system early.
* Development can be divided into smaller parts and more risky parts can be developed earlier which helps better risk management.

However, the model was rejected on account of the following negatives associated with the model:

* Management is more complex.
* End of project may not be known early.
* Not suitable for small or low risk projects and could be expensive for small projects.
* Spiral may go indefinitely.
* Large number of intermediate stages requires excessive documentation.

The Rapid Application Development (RAD) model is based on prototyping and iterative development with no specific planning involved. The process of writing the software itself involves the planning required for developing the product. While the model incorporates many of the features desired by the project team, such as:

* Changing requirements can be accommodated.
* Progress can be measured.
* Iteration time can be short with use of powerful RAD tools.
* Productivity with fewer people in short time.
* Reduced development time.
* Increased reusability of components.
* Quick initial reviews occur.
* Encourages customer feedback.
* Integration from very beginning solves a lot of integration issues.

since it discards planning in favour of rapid prototyping, it fails to provide the reliability the team was looking for in a process model. Moreover, it requires technically strong team members for identifying business requirements and highly skilled developers/designers. Also, if one fails to integrate the modules from the very beginning, severe compatibility issues can surface later on.

## 

## References:

* Software Development Life Cycle (SDLC), tutorialspoint.com.
* <http://en.wikipedia.org/>