THE PROTOTYPING MODEL

Often, a customer defines a set of general objectives for software but does not identify detailed input, processing, or output requirements. In other cases, the developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system, or the form that human/machine interaction should take. In these, and many other situations, a ***prototyping paradigm*** may offer the best approach.

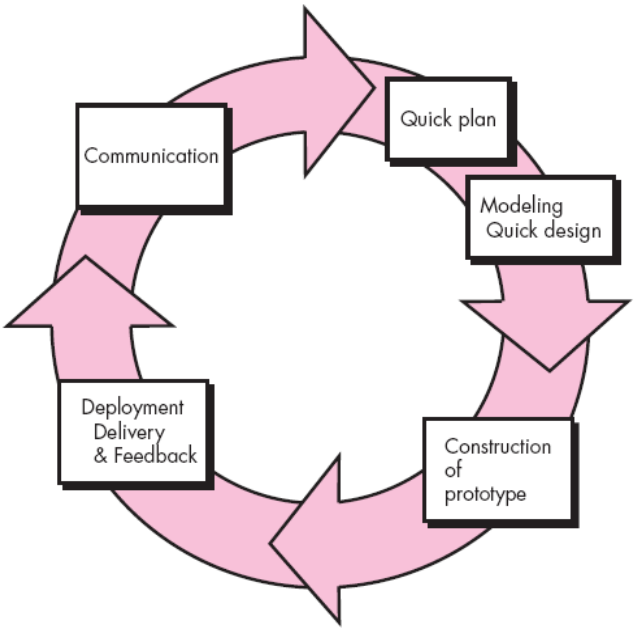
The prototyping paradigm begins with requirements gathering. Developer and customer meet and define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is Mandatory. A "quick design" then occurs. The quick design focuses on a representation of those aspects of the software that will be visible to the customer/user (e.g., input approaches and output formats). The quick design leads to the construction of a prototype.

The prototype is evaluated by the customer/user and used to refine requirements for the software to be developed. Iteration occurs as the prototype is tuned to satisfy the needs of the customer, while at the same time enabling the developer to better understand what needs to be done.

Ideally, the prototype serves as a mechanism for identifying software requirements. The prototype can serve as "the first system." The one that we throw away.

Prototyping model is a process of making software that is repetitive and with rapid planning where there is feedback that allows the occurrence of

repetition and improvement of software until the software meets the needs of the user.



Advantages

- **Improved requirements** −Prototypes allow customers to see what the finished application will look like. That lets them provide feedback to modify the requirements early in the project. Often customers can spot problems and request changes earlier so the finished result is more useful to users.

- **Common vision** −Prototypes let the customers and developers see the same preview of the finished application, so they are more likely to have a common vision of what the application should do and what it should look like.

- **Better design** −Prototypes let the developers quickly explore specific pieces of the application to learn what they involve. Prototypes also let developers test different approaches to see which one is best. The developers can use what they learn from the prototypes to improve the design and make the final code more elegant and robust.

Disadvantages

- **Narrowing vision** −People tend to focus on a prototype's specific approach rather than on the problem it addresses. When you show customers (and developers) a prototype, they will be less likely to think about other solutions that might do a better job.

  To avoid this problem, either don't build a prototype until you've considered possible alternatives, or build several prototypes to choose from.

- **Customer impatience** −A good prototype can make customers think that the finished application is just around the corner. They’ ll say things like, “The prototype looks good. Can't you just add a little error handling and a few extra features and call it done?”

  To avoid this, make sure customers realize that the prototype isn't anywhere close to the finished application.

- **Schedule pressure** −This goes with the preceding issue. If customers see a prototype that they think is mostly done, they may not understand that

you need another year to finish and may pressure you to shorten the schedule.

To avoid this problem, the project manager, executive champion, and other management types need to manage customer expectations so that they know what will be ready and when.

- **Raised expectations** −Sometimes, a prototype may demonstrate features that won't be included in the application. For example, those features might turn out to be too hard. Sometimes, features are included to assess their value to users, and the features are dropped if they don't have enough benefit. Other times a feature may be present just to show a possible future direction. In those cases, users may be disappointed when their favorite features are missing from the finished application. This can be a particular problem with projects that release a series of versions of the application and someone's pet feature isn't included in release 1.0.

  To avoid this, make sure customers understand which features will be included and when.

- **Attachment to code** −Sometimes, developers become attached to the prototype's code. That can make them follow the methods used by that code (or even reuse the code wholesale) even if a better design exists. This can be a particularly bad problem with throwaway prototypes where the initial code might have low quality.

  To avoid this, make sure developers understand that the code should change if a better design is available. Hold design reviews and code

reviews to make sure no one is stuck following a prototype approach if there's a better alternative.

- **Never - ending prototypes** −Throwaway prototypes are supposed to be built relatively quickly to provide fast feedback. Sometimes, developers spend far too much time refining a prototype to make it look better and include more features that aren’ t actually necessary.

  To avoid this, make sure the prototype doesn't include any more than is necessary to give customers a feel for how the program will work. Don't waste time making the prototype more flexible than necessary. Do the least amount of work you can get away with. For example, a prototype rarely needs to use a database. Usually you can just hard - wire data into the program to get a feel for how the final program will look. If customers decide they need to see more, they can say so.

Prototypes are great for helping you decide on the direction you should take. They can help define the user interface and other features, make sure customers and developers are on the same page, and let developers explore different solutions.