Software prototyping

**Software prototyping** is the activity of creating **prototypes** of software applications, i.e., incomplete versions of the **software program** being developed. It is an activity that can occur in **software development** and is comparable to **prototyping** as known from other fields, such as **mechanical engineering** or **manufacturing**.

A prototype typically simulates only a few aspects of, and may be completely different from, the final product.

Prototyping has several benefits: 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. The degree of completeness and the techniques used in the prototyping have been in development and debate since its proposal in the early 1970s.

**Overview**

The original purpose of a prototype is 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. Prototyping can also be used by end users to describe and prove requirements that have not been considered, and that can be a key factor in the commercial relationship between developers and their clients.[1] **Interaction design** in particular makes heavy use of prototyping with that goal.

This process is in contrast with the 1960s and 1970s monolithic development cycle of building the entire program first and then working out any inconsistencies between design and implementation, which led to higher software costs and poor estimates of time and cost. [citation needed] The monolithic approach has been dubbed the "Slaying the (software) Dragon" technique, since it assumes that the software designer and developer is a single hero who has to slay the entire dragon alone. Prototyping can also avoid the great expense and difficulty of changing a finished software product.

The practice of prototyping is one of the points Frederick P. Brooks makes in his 1975 book *The Mythical Man-Month* and his 10-year anniversary article *No Silver Bullet*.

An early example of large-scale software prototyping was the implementation of NYU's Ada/ED translator for the **Ada programming language** [2]. It was implemented in **SETL** with the intent of producing an executable semantic model for the Ada language, emphasizing clarity of design and user interface over speed and efficiency. The NYU Ada/ED system was the first validated Ada implementation, certified on April 11, 1983.[3]
Outline of the prototyping process

The process of prototyping involves the following steps

1. Identify basic requirements

   Determine basic requirements including the input and output information desired. Details, such as security, can typically be ignored.

2. Develop Initial Prototype

   The initial prototype is developed that includes only user interfaces. (See Horizontal Prototype, below)

3. Review

   The customers, including end-users, examine the prototype and provide feedback on additions or changes.

4. Revise and Enhance the Prototype

   Using the feedback both the specifications and the prototype can be improved. Negotiation about what is within the scope of the contract/product may be necessary. If changes are introduced then a repeat of steps #3 and #4 may be needed.

Dimensions of prototypes

Nielsen summarizes the various dimensions of prototypes in his book Usability Engineering

Horizontal Prototype

A common term for a user interface prototype is the horizontal prototype. It provides a broad view of an entire system or subsystem, focusing on user interaction more than low-level system functionality, such as database access. Horizontal prototypes are useful for:

- Confirmation of user interface requirements and system scope
- Demonstration version of the system to obtain buy-in from the business
- Develop preliminary estimates of development time, cost and effort

Vertical Prototype

A vertical prototype is a more complete elaboration of a single subsystem or function. It is useful for obtaining detailed requirements for a given function, with the following benefits:
Software prototyping has many variants. However, all the methods are in some way based on two major types of prototyping: Throwaway Prototyping and Evolutionary Prototyping.

**Throwaway prototyping**

Also called close-ended prototyping. Throwaway or Rapid Prototyping refers to the creation of a model that will eventually be discarded rather than becoming part of the final delivered software. After preliminary requirements gathering is accomplished, a simple working model of the system is constructed to visually show the users what their requirements may look like when they are implemented into a finished system.

Rapid Prototyping involved creating a working model of various parts of the system at a very early stage, after a relatively short investigation. The method used in building it is usually quite informal, the most important factor being the speed with which the model is provided. The model then becomes the starting point from which users can re-examine their expectations and clarify their requirements. When this has been achieved, the prototype model is 'thrown away', and the system is formally developed based on the identified requirements.

The most obvious reason for using Throwaway Prototyping is that it can be done quickly. If the users can get quick feedback on their requirements, they may be able to refine them early in the development of the software. Making changes early in the development lifecycle is extremely cost effective since there is nothing at that point to redo. If a project is changed after a considerable work has been done then small changes could require large efforts to implement since software systems have many dependencies. Speed is crucial in implementing a throwaway prototype, since with a limited budget of time and money little can be expended on a prototype that will be discarded.

Another strength of Throwaway Prototyping is its ability to construct interfaces that the users can test. The user interface is what the user sees as the system, and by seeing it in front of them, it is much easier to grasp how the system will work.

...it is asserted that revolutionary **rapid prototyping** is a more effective manner in which to deal with user requirements-related issues, and therefore a greater enhancement to software productivity overall. Requirements can be identified, simulated, and tested far more quickly and cheaply when issues of evolvability, maintainability, and software structure are ignored. This, in turn, leads to the accurate specification of requirements, and the subsequent construction of a valid and usable system from the user's perspective via conventional software development models.
Prototypes can be classified according to the fidelity with which they resemble the actual product in terms of appearance, interaction and timing. One method of creating a low fidelity Throwaway Prototype is **Paper Prototyping**. The prototype is implemented using paper and pencil, and thus mimics the function of the actual product, but does not look at all like it. Another method to easily build high fidelity Throwaway Prototypes is to use a **GUI Builder** and create a *click dummy*, a prototype that looks like the goal system, but does not provide any functionality.

Not exactly the same as Throwaway Prototyping, but certainly in the same family, is the usage of **storyboards**, animatics or drawings. These are non-functional implementations but show how the system will look.

Summary: In this approach the prototype is constructed with the idea that it will be discarded and the final system will be built from scratch. The steps in this approach are:

1. Write preliminary requirements
2. Design the prototype
3. User experiences/uses the prototype, specifies new requirements
4. Repeat if necessary
5. Write the final requirements

**Evolutionary prototyping**

Evolutionary Prototyping (also known as **breadboard prototyping**) is quite different from Throwaway Prototyping. The main goal when using Evolutionary Prototyping is to build a very robust prototype in a structured manner and constantly refine it. The reason for this is that the Evolutionary prototype, when built, forms the heart of the new system, and the improvements and further requirements will be built.

When developing a system using Evolutionary Prototyping, the system is continually refined and rebuilt.

"...evolutionary prototyping acknowledges that we do not understand all the requirements and builds only those that are well understood."[5]

This technique allows the development team to add features, or make changes that couldn't be conceived during the requirements and design phase.

For a system to be useful, it must evolve through use in its intended operational environment. A product is never "done;" it is always maturing as the usage environment changes...we often try to define a system using our most familiar frame of reference---where we are now. We make assumptions about the way business will be conducted and the technology base on which the business will be implemented. A plan is enacted to develop the capability, and, sooner or later, something resembling the envisioned system is delivered.[9]
Evolutionary Prototypes have an advantage over Throwaway Prototypes in that they are functional systems. Although they may not have all the features the users have planned, they may be used on an interim basis until the final system is delivered.

"It is not unusual within a prototyping environment for the user to put an initial prototype to practical use while waiting for a more developed version... The user may decide that a 'flawed' system is better than no system at all."[7]

In Evolutionary Prototyping, developers can focus themselves to develop parts of the system that they understand instead of working on developing a whole system.

To minimize risk, the developer does not implement poorly understood features. The partial system is sent to customer sites. As users work with the system, they detect opportunities for new features and give requests for these features to developers. Developers then take these enhancement requests along with their own and use sound configuration-management practices to change the software-requirements specification, update the design, recode and retest.[10]

Incremental prototyping

The final product is built as separate prototypes. At the end the separate prototypes are merged in an overall design. By the help of incremental prototyping we can reduce the time gap between user and software developer.

Extreme prototyping

Extreme Prototyping as a development process is used especially for developing web applications. Basically, it breaks down web development into three phases, each one based on the preceding one. The first phase is a static prototype that consists mainly of HTML pages. In the second phase, the screens are programmed and fully functional using a simulated services layer. In the third phase, the services are implemented. The process is called Extreme Prototyping to draw attention to the second phase of the process, where a fully functional UI is developed with very little regard to the services other than their contract.

Advantages of prototyping

There are many advantages to using prototyping in software development – some tangible, some abstract.[11]

**Reduced time and costs**: Prototyping can improve the quality of requirements and specifications provided to developers. Because changes cost exponentially more to implement as they are detected later in development, the early determination of *what the user really wants* can result in faster and less expensive software.[8]
Improved and increased user involvement: Prototyping requires user involvement and allows them to see and interact with a prototype allowing them to provide better and more complete feedback and specifications. The presence of the prototype being examined by the user prevents many misunderstandings and miscommunications that occur when each side believe the other understands what they said. Since users know the problem domain better than anyone on the development team does, increased interaction can result in final product that has greater tangible and intangible quality. The final product is more likely to satisfy the users desire for look, feel and performance.

Disadvantages of prototyping

Using, or perhaps misusing, prototyping can also have disadvantages.

Insufficient analysis: The focus on a limited prototype can distract developers from properly analyzing the complete project. This can lead to overlooking better solutions, preparation of incomplete specifications or the conversion of limited prototypes into poorly engineered final projects that are hard to maintain. Further, since a prototype is limited in functionality it may not scale well if the prototype is used as the basis of a final deliverable, which may not be noticed if developers are too focused on building a prototype as a model.

User confusion of prototype and finished system: Users can begin to think that a prototype, intended to be thrown away, is actually a final system that merely needs to be finished or polished. (They are, for example, often unaware of the effort needed to add error-checking and security features which a prototype may not have.) This can lead them to expect the prototype to accurately model the performance of the final system when this is not the intent of the developers. Users can also become attached to features that were included in a prototype for consideration and then removed from the specification for a final system. If users are able to require all proposed features be included in the final system this can lead to conflict.

Developer misunderstanding of user objectives: Developers may assume that users share their objectives (e.g. to deliver core functionality on time and within budget), without understanding wider commercial issues. For example, user representatives attending Enterprise software (e.g. PeopleSoft) events may have seen demonstrations of “transaction auditing” (where changes are logged and displayed in a difference grid view) without being told that this feature demands additional coding and often requires more hardware to handle extra database accesses. Users might believe they can demand auditing on every field, whereas developers might think this is feature creep because they have made assumptions about the extent of user requirements. If the developer has committed delivery before the user requirements were reviewed, developers are between a rock and a hard place, particularly if user management derives some advantage from their failure to implement requirements.

Developer attachment to prototype: Developers can also become attached to prototypes they have spent a great deal of effort producing; this can lead to problems like attempting to convert a limited prototype into a final system when it does not have an appropriate underlying architecture. (This may suggest that throwaway prototyping, rather than evolutionary prototyping, should be used.)
Excessive development time of the prototype: A key property to prototyping is the fact that it is supposed to be done quickly. If the developers lose sight of this fact, they very well may try to develop a prototype that is too complex. When the prototype is thrown away the precisely developed requirements that it provides may not yield a sufficient increase in productivity to make up for the time spent developing the prototype. Users can become stuck in debates over details of the prototype, holding up the development team and delaying the final product.

Expense of implementing prototyping: the start up costs for building a development team focused on prototyping may be high. Many companies have development methodologies in place, and changing them can mean retraining, retooling, or both. Many companies tend to just jump into the prototyping without bothering to retrain their workers as much as they should.

A common problem with adopting prototyping technology is high expectations for productivity with insufficient effort behind the learning curve. In addition to training for the use of a prototyping technique, there is an often overlooked need for developing corporate and project specific underlying structure to support the technology. When this underlying structure is omitted, lower productivity can often result. [13]

Best projects to use prototyping

It has been argued that prototyping, in some form or another, should be used all the time. However, prototyping is most beneficial in systems that will have many interactions with the users.

It has been found that prototyping is very effective in the analysis and design of on-line systems, especially for transaction processing, where the use of screen dialogs is much more in evidence. The greater the interaction between the computer and the user, the greater the benefit is that can be obtained from building a quick system and letting the user play with it. [7]

Systems with little user interaction, such as batch processing or systems that mostly do calculations, benefit little from prototyping. Sometimes, the coding needed to perform the system functions may be too intensive and the potential gains that prototyping could provide are too small. [7]

Prototyping is especially good for designing good human-computer interfaces. "One of the most productive uses of rapid prototyping to date has been as a tool for iterative user requirements engineering and human-computer interface design." [8]