Rapid Prototyping in Software Development

Karl Soop
2004-02-15

Basics

Rapid Prototyping is a technique for software development that has been successfully adapted by many leading-edge development laboratories during the last 10 years or so. Its advantages are:

- Designers, decision makers, and potential users get a quick look and feel for the planned product, long before it has become a reality. This lays the ground for realistic project plans.
- The prototype is used as an extremely efficient vehicle for presenting and experimenting with design ideas. It becomes an evolving component in the project, potentially merging into or even transforming into production code.
- The prototype may be used in lieu of basic documentation, and it may form the base for more evolved documents.

How does it work?

Let's assume that there is, within the lab organisation, some sort of *process* that projects are supposed to follow. Usually the process describes the various phases of a project and the roles of various project members. Rapid Prototyping heavily influences this process. We shall continue speaking about *phases*, even if they are not necessarily discreet in time.

A development process usually recognises these phases: requirements gathering, analysis and design, implementation (coding), testing, shipping, and maintenance. Several alternative terms are used, more or less equivalently. Let's for the moment look at the first two phases.

Before a new project is begun there usually exists at least a rough idea of product requirements, which have been documented. Ideally, users have been interviewed, experience from earlier projects reused, and requirements gathered from authoritative sources.

Now for the analysis phase, a project member begins by putting together some chunks of code, usually from existing class libraries, into a very embryonic prototype. Alternatively, one may resort to some specific prototyping tool. This is usually much faster than having meetings and writing specs. Meetings and program specifications are typically part of what we might call the *Traditional Approach*.

The early, "embryonic" prototype should be put together by one senior developer with extensive experience in Object-Oriented (*OO*) design, e.g. using the *rup* methodology. As the project expands, more people will get involved, and the prototype becomes increasingly important as a discussion vehicle and as a source of documentation.

The Prototype in the Project

The material used for the initial prototype is preferably borrowed from earlier projects that exhibit some traits in common with the new product. Usually some common class libraries can be reused.

If we are dealing with a new version of an existing product, this is easy. If no suitable code exists, there is some extra work involved in this phase. Note, however, that at this stage the accent is on user interfaces and basic functionality. For example, data-base accesses, communication with other environments, and complex algorithms may be faked or bypassed.

The initial prototype plays a central role in meetings and project planning. At the first design meeting, the prototype will be demoed to the participants, who will be able to interact directly with it and discuss changes and improvements. For example, issues on suitable user navigation can be resolved early on, simply by actually navigating. Later versions of the prototype become the corner-stone of all subsequent design meetings. The early prototype is a first cut of the future product. Given suitable prototyping tools, it ideally *evolves into production code* by stepwise refinement.

The prototype is also used in management and planning meetings as the central vehicle for realistic time and resource estimates. It is even possible to delay project kick-off (i.e. decide whether to launch the project at all) until the prototype has been reviewed after one or a few iterations. The first *real project plan* is based on the prototype, it is therefore much more realistic than traditional project planning. In fact this is the *only* way to make realistic plans. Estimates building on the Traditional Approach (i.e. code-line counting) are notoriously inaccurate, and invariably result in project cost overruns (see below).

As far as functional specifications go, the prototype is its own specification in the sense that very little additional prose is needed, provided the prototype is shipped along to decision-makers, reviewers, and sign-offs in the process. Finally, the prototype may evolve into a tutorial to be integrated into the product.

A direct result of the above is that meetings, planning, and internal documentation are kept to a minimum, freeing up resources for productive tasks.

On Prototyping Languages and Tools

It is desirable that the language used to express the prototype, or generated by prototyping activities, be the same programming language as will be used in implementation. If possible, one should avoid special prototyping tools, as these tend to impose their own idiosyncratic and often unwanted conventions on the design and coding phases. Moreover, they tend to change with the flavour of the month.

Often the management forces various prototyping and development tools ("environments") on the developers simply because they are perceived as "strategic" (see below). Such tools tend to cater to all phases in the process, and consequently become too big, too unwieldy, and take too much space. They try to solve too many problems, cramming too much function into one package (the *monolith syndrome*). They take days to install, months to learn, and come with thousands of pages of documentation[1]. They clog servers and workstations with gigabytes of code, and execute at snail's pace. Even the final code shipped with the product suffers a penalty by expanding beyond reasonable limits. Programmers don't like this. Instead programmers like small, nifty tools that make one thing only and do it right. Examples are small, special-purpose editors (e.g. dialog editors) and graphical OO-design packages.

Despite all this, the development process should be open to the choice between a special prototyping tool and sticking to the programming language. It should be clear that using a

[1]If the first title of the book says: "How to use this manual", just junk the whole lot.

programming language is contingent upon the existence of suitable class libraries to pick large chunks from. A "class library" will sometimes be merely some code that can be reused and modified from an earlier project (this may indeed be the most likely start for an early prototype). If a programming language is used, the prototype should evolve into production code, as said above, and it is a matter of taste when one wants to stop calling it "prototype". In this case, *cutover* would probably happen when the prototype gets updated after the first design meeting.

On Iterative Development

The Traditional Approach upholds old-style linear development, the so-called the "waterfall method", or "one thing after the other". This is the manager's and the project planner's sweet dream: each phase ends at a pre-determined time, having consumed pre-determined resources, before the new phase is entered. Earlier phases need never be revisited.

As all developers know, this never works in practice. Virtually no project lives up to waterfall expectations. Typically, a successful development project requires many iterations back to previous phases, with cycles varying from a week up to several months duration. Iterating coding and unit testing is hardly controversial, but practicality often demands the developer also to revisit prototyping. Smaller units will be re-designed along with some usability testing in a cyclic mode, as the work progresses. On the documentation side, design and even functional specifications will need to be updated collaterally on a cyclic basis.

The iterative method of progressing a development project is more in line with modern thinking, and in particular with OO principles. In its extreme, the method may even lead to updates of requirements and functional objectives. For example, if it becomes apparent that fixing a problem is easier if a function is generalised, rather than sticking to a restrictive specification, the fix usually impacts overall functionality, user interface, and documentation.

Abandoning the waterfall method entails several advantages:

- The total development period tends to be shorter, meaning lower cost and improved product timeliness. This is because problems are detected earlier in the process, and their fixing may require — usually small — updates to the design. Leaving the initial design intact, more often than not forces the development team to late design fixes resulting in major restructuring of code, test, and documentation — an expensive alternative.

- Product maintainability tends to be higher as changes are integrated early in the process. Late changes are likely to exhibit the "spaghetti" syndrome, leading to higher cost in the post-production service process.

- Quality is likely to be higher. Apart from being easier to maintain, the product tends to be more robust, leading to higher customer satisfaction.

On Project Organisation

There are small and there are large projects, and management likes to believe that the larger the project the more workers are needed. But as is well-known (since Parkinson) too many project members inhibit productivity. Adding more people above a certain threshold actually slows down the project, causing it to cost more at no benefit. The reason is not difficult to glean: people tend to get into each other's way and bureaucracy and planning overhead eat the rest of potential gains.

So what is a suitable threshold? In my opinion, a development team, adhering to the modern principles expressed in this document, would typically consist of 1-5 members, on a large project perhaps up to a maximum of 7. This is a far cry from the traditional "sweat-shops" or "programming factories" of the 1970s, where literally hundreds of programmers worked on the same project in a waterfall fashion. If the project is too big for the recommended small team, one should seek to split it into sub-projects, each with its precisely defined goal. On the other hand, one might ask oneself: isn't such a large project perhaps a mistake anyway? Witness the high number of costly failures, invariably involving huge projects with huge cost overruns (cf. the Swedish public-sector undertakings during the last decade).

How to measure the "size" of a project? For planning purposes, the Traditional Approach is to estimate the number of lines of code, in other words, pure guesswork. A development effort depends on many activities, not factored into lines-of-code.

First, there is a one-time consumption of resources and elapsed time in each project. This includes learning what the product is supposed to do, possibly also getting familiar with other systems that the product is supposed to interact with, including earlier versions of the same product.

In addition, the "flavour-of-the-month" syndrome is rampant in many labs. Management is fond of imposing *new tools and new platforms* on developers, just because "now we are supposed to use this new marvelous version of X and everything will be much more productive", insisting that X is "*strategic*". This is a fallacy:

a. X *never* works correctly, since it was developed only a month ago by vendor Y and we are getting a beta version;

b. Y has not yet documented X, or only provides an inaccurate, sloppy, and incomplete documentation;

c. X, when the sharp version finally arrives, will have changed so much that we have to adapt our code at a great cost;

d. Using X is hazardous, causing repeated crashes in the beginning of the project, and engenders multiple delays, not accounted for in the plan.

e. If you run into problems, Y has either gone bankrupt or reorganised, so no support is forthcoming.

If programmers are allowed to develop solely with their own experience and knowledge, relying on well-proven and familiar tools, they are likely to produce robust and well-functioning code in a productive way in a timely fashion. Education with the new tool should be sought in-between projects, not in their midst.

On User Interfaces

The *user interface* is what the ultimate user will see and experience when running our product. The user interface is not only crucial for user satisfaction: it is a *major competitive factor* in the marketplace. The user interface is the window to the outside when the product is demoed to customers, in seminars, in trade fairs. It is the key to your reputation. It may be said to equal, if not surpass, *functionality* when it comes to product rating.

Focus should therefore be firmly set on the user interface throughout the whole development process. If you cannot afford to invest in professional interface design, you might just as well cancel the project. The main principle is this:

The user interface must be seen as an integrated part of the product, and as such its design, implementation, and test, should be an integrated part of the development process. The accent is here on *integrated.*

With the Traditional Approach, design of a user interface is assigned externally to the development process. And, even if this is not the intent, the process often tends to get crippled by an entirely artificial split that removes interface designers from the development group. Interface design is then brought in only on an as-needed basis, typically at the end of the project to touch up the cluttered results of the programmers' attempt. This leads to the traditional failure of the product in the marketplace, due to mediocre human factors, sloppy and counter-intuitive interfaces, inconsistent navigation, and a drab, unsellable look.

Interface design involves user-interface prototyping, human-factors tradeoff and assessment, ensuring compliance to standards, and usability testing. It fits excellently with Rapid Prototyping.

Interface design can only be successfully undertaken by experts with the required education and skills, skills that are usually not found with traditional programmers or systems engineers. To ensure quality, these experts must participate in *all* phases of the development process: in requirements gathering, in design, implementation, documentation, testing, and maintenance.

User interfaces may be split into:

1. documentation (including on-line); and
2. application interface (e.g. GUI).

While both parts contribute in fundamental ways to the final product, they are, however, of a totally different technical nature. On-line documentation (including help and tutorials) is largely managed by built-in facilities of the operating system, so the expert mainly concentrates on developing text spanned by a navigational network.

The application interface, on the other hand, is intimately interwoven with a program logic that must respond to a wide variety of user interactions in a large number of application states. The user interface must therefore be designed and implemented together with the remainder of the code, requiring the implementer and the interface expert to work hand in hand. It is evident that this goal can be achieved only if the interface-design experts are fully-fledged members of the development team, reporting to the Project Manager.

On Project Documentation

Under this heading, we consider primarily internal documentation used as reference material by the project team members for the duration of the project. It is a well-known fact that programmers hate to document.

We should start by observing that there will always be *some* minimal documentation on paper. With Rapid Prototyping, however, this part is more or less limited to product *requirements.* Design documentation is preferably in the form of OO diagrams, i.e. the output from an Analysis tool. As regards logic design, the class definitions (header files), if adequately documented, will provide most of what is needed. Lastly, the on-line help and (where available) tutorial, provides most of the functional specifications. In summary, actual documentary prose is kept to its bare minimum with Rapid Prototyping.

Conclusion

The techniques described in this paper as Rapid Prototyping has proven beneficial to software development over the last decade or so, by (a) reducing product cost, and (b) enhancing product quality. It fits excellently with other proven technologies, such as structured programming, object-oriented design, and unified processes.