Software prototyping

From Wikipedia, the free encyclopedia

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 prototyping have been in development and debate since its proposal in the early 1970s.[6]

Contents

- 1 Overview
- 2 Outline of the prototyping process
- 3 Dimensions of prototypes
  - 3.1 Horizontal prototype
  - 3.2 Vertical prototype
- 4 Types of prototyping
  - 4.1 Throwaway prototyping
  - 4.2 Evolutionary prototyping
  - 4.3 Incremental prototyping
  - 4.4 Extreme prototyping
- 5 Advantages of prototyping
- 6 Disadvantages of prototyping
- 7 Best projects to use prototyping
  - 7.1 Dynamic systems development method
  - 7.2 Operational prototyping
  - 7.3 Evolutionary systems development

- 7.4 Evolutionary rapid development
- 8 Tools
  - 8.1 Screen generators, design tools, and software factories
  - 8.2 Application definition or simulation software
  - 8.3 Requirements Engineering Environment
  - 8.4 LYMB
  - 8.5 Non-relational environments
  - 8.6 PSDL
- 9 See also
- 10 Notes
- 11 References

Overview

The 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. 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 having to change 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 potential 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:

- Refinement database design,
- Obtain information on data volumes and system interface needs, for network sizing and performance engineering,
- Clarify complex requirements by drilling down to actual system functionality.

Types of prototyping

Software prototyping has many variants. However, all of the methods are in some way based on two major forms 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. It is also a rapid prototyping.

> Rapid prototyping involves 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 goal has been achieved, the prototype model is 'thrown away', and the system is formally developed based on the identified requirements.[7]

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 amount of 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 function.

> …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. [8]

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.

The usage of storyboards, animatics or drawings is not exactly the same as throwaway prototyping, but certainly falls within the same family. 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 approach is that the evolutionary prototype, when built, forms the heart of the new system, and the improvements and further requirements will then 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 the time gap between user and software developer is reduced.

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.[7] 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 a final product that has greater tangible and intangible quality. The final product is more likely to satisfy the user's 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, such as 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 begin 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]

Dynamic systems development method

Dynamic Systems Development Method (DSDM)[18] is a framework for delivering business solutions that relies heavily upon prototyping as a core technique, and is itself ISO 9001 approved. It expands upon most understood definitions of a prototype. According to DSDM the prototype may be a diagram, a business process, or even a system placed into production. DSDM prototypes are intended to be incremental, evolving from simple forms into more comprehensive ones.

DSDM prototypes can sometimes be *throwaway* or *evolutionary*. Evolutionary prototypes may be evolved horizontally (breadth then depth) or vertically (each section is built in detail with additional iterations detailing subsequent sections). Evolutionary prototypes can eventually evolve into final systems.

The four categories of prototypes as recommended by DSDM are:

- **Business prototypes** – used to design and demonstrates the business processes being automated.
- **Usability prototypes** – used to define, refine, and demonstrate user interface design usability, accessibility, look and feel.
- **Performance and capacity prototypes** - used to define, demonstrate, and predict how systems will perform under peak loads as well as to demonstrate and evaluate other non-functional aspects of the system (transaction rates, data storage volume, response time, etc.)
- **Capability/technique prototypes** – used to develop, demonstrate, and evaluate a design approach or concept.

The DSDM lifecycle of a prototype is to:

1. Identify prototype
2. Agree to a plan
3. Create the prototype
4. Review the prototype

Operational prototyping

Operational prototyping was proposed by Alan Davis as a way to integrate throwaway and evolutionary prototyping with conventional system development. "It offers the best of both the quick-and-dirty and conventional-development worlds in a sensible manner. Designers develop only well-understood features in building the evolutionary baseline, while using throwaway prototyping to experiment with the poorly understood features."[5]

Davis' belief is that to try to "retrofit quality onto a rapid prototype" is not the correct method when trying to combine the two approaches. His idea is to engage in an evolutionary prototyping methodology and rapidly prototype the features of the system after each evolution.

The specific methodology follows these steps: [5]

- An evolutionary prototype is constructed and made into a baseline using conventional development strategies, specifying and implementing only the requirements that are well understood.
- Copies of the baseline are sent to multiple customer sites along with a trained prototyper.
- At each site, the prototyper watches the user at the system.
- Whenever the user encounters a problem or thinks of a new feature or requirement, the prototyper logs it. This frees the user from having to record the problem, and allows him to continue working.
- After the user session is over, the prototyper constructs a throwaway prototype on top of the baseline system.
- The user now uses the new system and evaluates. If the new changes aren't effective, the prototyper removes them.
- If the user likes the changes, the prototyper writes feature-enhancement requests and forwards them to the development team.
- The development team, with the change requests in hand from all the sites, then produce a new evolutionary prototype using conventional methods.

Obviously, a key to this method is to have well trained prototypers available to go to the user sites. The operational prototyping methodology has many benefits in systems that are complex and have few known requirements in advance.

Evolutionary systems development

Evolutionary Systems Development is a class of methodologies that attempt to formally implement evolutionary prototyping. One particular type, called Systemscraft is described by John Crinnion in his book *Evolutionary Systems Development*.

Systemscraft was designed as a 'prototype' methodology that should be modified and adapted to fit the specific environment in which it was implemented.

> Systemscraft was not designed as a rigid 'cookbook' approach to the development process. It is now generally recognised[sic] that a good methodology should be flexible enough to be adjustable to suit all kinds of environment and situation...[7]

The basis of Systemscraft, not unlike evolutionary prototyping, is to create a working system from the initial requirements and build upon it in a series of revisions. Systemscraft places heavy emphasis on traditional analysis being used throughout the development of the system.

Evolutionary rapid development

Evolutionary Rapid Development (ERD)[12] was developed by the Software Productivity Consortium, a technology development and integration agent for the Information Technology Office of the Defense Advanced Research Projects Agency (DARPA).

> Fundamental to ERD is the concept of composing software systems based on the reuse of components, the use of software templates and on an architectural template. Continuous evolution of system capabilities in rapid response to changing user needs and technology is highlighted by the evolvable architecture, representing a class of solutions. The process focuses on the use of small artisan-based teams integrating software and systems engineering disciplines working multiple, often parallel short-duration timeboxes with frequent customer interaction.

> Key to the success of the ERD-based projects is parallel exploratory analysis and development of features, infrastructures, and components with and adoption of leading edge technologies enabling the quick reaction to changes in technologies, the marketplace, or customer requirements.[9]

To elicit customer/user input, frequent scheduled and ad hoc/impromptu meetings with the stakeholders are held. Demonstrations of system capabilities are held to solicit feedback before design/implementation decisions are solidified. Frequent releases (e.g., betas) are made available for use to provide insight into how the system could better support user and customer needs. This assures that the system evolves to satisfy existing user needs.

The design framework for the system is based on using existing published or de facto standards. The system is organized to allow for evolving a set of capabilities that includes considerations for performance, capacities, and functionality. The architecture is defined in terms of abstract interfaces that encapsulate the services and their implementation (e.g., COTS applications). The architecture serves as a template to be used for guiding development of more than a single instance of the system. It allows for multiple application components to be used to implement the services. A core set of functionality not likely to change is also identified and established.

The ERD process is structured to use demonstrated functionality rather than paper products as a way for stakeholders to communicate their needs and expectations. Central to this goal of rapid delivery is the use of the "timebox" method. Timeboxes are fixed periods of time in which specific tasks (e.g., developing a set of functionality) must be performed. Rather than allowing time to expand to satisfy some vague set of goals, the time is fixed (both in terms of calendar weeks and person-hours) and a set of goals is defined that realistically can be achieved within these constraints. To keep development from degenerating into a "random walk," long-range plans are defined to guide the iterations. These plans provide a vision for the overall system and set boundaries (e.g., constraints) for the project. Each iteration within the process is conducted in the context of these long-range plans.

Once an architecture is established, software is integrated and tested on a daily basis. This allows the team to assess progress objectively and identify potential problems quickly. Since small amounts of the system are integrated at one time, diagnosing and removing the defect is rapid. User demonstrations can be held at short notice since the system is generally ready to exercise at all times.

Tools

Efficiently using prototyping requires that an organization have the proper tools and a staff trained to use those tools. Tools used in prototyping can vary from individual tools, such as 4th generation programming languages used for rapid prototyping to complex integrated CASE tools. 4th generation visual programming languages like Visual Basic and ColdFusion are frequently used since they are cheap, well known and relatively easy and fast to use. CASE tools, supporting requirements analysis, like the Requirements Engineering Environment (see below) are often developed or selected by the military or large organizations. Object oriented tools are also being developed like LYMB from the GE Research and Development Center. Users may prototype elements of an application themselves in a spreadsheet.

As web-based applications continue to grow in popularity, so too, have the tools for prototyping such applications. Frameworks such as Bootstrap, Foundation, and AngularJS provide the tools necessary to quickly structure a proof of concept. These frameworks typically consist of a set of controls, interactions, and design guidelines that enable developers to quickly prototype web applications.

Screen generators, design tools, and software factories

Screen generating programs are also commonly used and they enable prototypers to show user's systems that do not function, but show what the screens may look like.[4] Developing Human Computer Interfaces can sometimes be the critical part of the development effort, since to the users the interface essentially is the system.

Software factories can generate code by combining ready-to-use modular components. This makes them ideal for prototyping applications, since this approach can quickly deliver programs with the desired behaviour, with a minimal amount of manual coding.

Application definition or simulation software

A new class of software called **Application definition or simulation software** enables users to rapidly build lightweight, animated simulations of another computer program, without writing code. Application simulation software allows both technical and non-technical users to experience, test, collaborate and validate the simulated program, and provides reports such as annotations, screenshot and schematics. As a solution specification technique, Application Simulation falls between low-risk, but limited, text or drawing-based mock-ups (or wireframes)

sometimes called *paper-based prototyping*, and time-consuming, high-risk code-based prototypes, allowing software professionals to validate requirements and design choices early on, before development begins. In doing so, the risks and costs associated with software implementations can be dramatically reduced.[5]

To simulate applications one can also use software that simulates real-world software programs for computer-based training, demonstration, and customer support, such as screencasting software as those areas are closely related. There are also more specialised tools.[6][7][8]

Requirements Engineering Environment

"The Requirements Engineering Environment (REE), under development at Rome Laboratory since 1985, provides an integrated toolset for rapidly representing, building, and executing models of critical aspects of complex systems."[15]

Requirements Engineering Environment is currently used by the United States Air Force to develop systems. It is:

> an integrated set of tools that allows systems analysts to rapidly build functional, user interface, and performance prototype models of system components. These modeling activities are performed to gain a greater understanding of complex systems and lessen the impact that inaccurate requirement specifications have on cost and scheduling during the system development process. Models can be constructed easily, and at varying levels of abstraction or granularity, depending on the specific behavioral aspects of the model being exercised.[15]

REE is composed of three parts. The first, called proto is a CASE tool specifically designed to support rapid prototyping. The second part is called the Rapid Interface Prototyping System or RIP, which is a collection of tools that facilitate the creation of user interfaces. The third part of REE is a user interface to RIP and proto that is graphical and intended to be easy to use.

Rome Laboratory, the developer of REE, intended that to support their internal requirements gathering methodology. Their method has three main parts:

- Elicitation from various sources (users, interfaces to other systems), specification, and consistency checking

- Analysis that the needs of diverse users taken together do not conflict and are technically and economically feasible
- Validation that requirements so derived are an accurate reflection of user needs.[15]

In 1996, Rome Labs contracted Software Productivity Solutions (SPS) to further enhance REE to create "a commercial quality REE that supports requirements specification, simulation, user interface prototyping, mapping of requirements to hardware architectures, and code generation..."[16] This system is named the Advanced Requirements Engineering Workstation or AREW.

LYMB

LYMB[17] is an object-oriented development environment aimed at developing applications that require combining graphics-based user interfaces, visualization, and rapid prototyping.

Non-relational environments

Non-relational definition of data (e.g. using Caché or associative models) can help make end-user prototyping more productive by delaying or avoiding the need to normalize data at every iteration of a simulation. This may yield earlier/greater clarity of business requirements, though it does not specifically confirm that requirements are technically and economically feasible in the target production system.

PSDL

PSDL is a prototype description language to describe real-time software.[9] The associated tool set is CAPS (Computer Aided Prototyping System).[10] Prototyping software systems with hard real-time requirements is challenging because timing constraints introduce implementation and hardware dependencies. PSDL addresses these issues by introducing control abstractions that include declarative timing constraints. CAPS uses this information to automatically generate code and associated real-time schedules, monitor timing constraints during prototype execution, and simulate execution in proportional real time relative to a set of parameterized hardware models. It also provides default assumptions that enable

execution of incomplete prototype descriptions, integrates prototype construction with a software reuse repository for rapidly realizing efficient implementations, and provides support for rapid evolution of requirements and designs.[11]

See also

- Comparison of software prototyping tools

Notes

1. **^** C. Melissa Mcclendon, Larry Regot, Gerri Akers: The Analysis and Prototyping of Effective Graphical User Interfaces. October 1996. [3] (http://www.umsl.edu/~s980548/gproj1/intro.html)
2. **^** D.A. Stacy, professor, University of Guelph. Guelph, Ontario. Lecture notes on Rapid Prototyping. August, 1997. [4] (http://hebb.cis.uoguelph.ca/~dave/343/Lectures/prototype.html)
3. **^** Stephen J. Andriole: Information System Design Principles for the 90s: Getting it Right. AFCEA International Press, Fairfax, Virginia. 1990. Page 13.
4. **^** R. Charette, Software Engineering Risk Analysis and Management. McGraw Hill, New York, 1989.
5. **^** Alan M. Davis: Operational Prototyping: A new Development Approach. IEEE Software, September 1992. Page 71.
6. **^** Todd Grimm: The Human Condition: A Justification for Rapid Prototyping. Time Compression Technologies, vol. 3 no. 3. Accelerated Technologies, Inc. May 1998 . Page 1. [5] (http://www.tagrimm.com/publications/art-human-1998.html)
7. **^** John Crinnion: Evolutionary Systems Development, a practical guide to the use of prototyping within a structured systems methodology. Plenum Press, New York, 1991. Page 18.
8. **^** S. P. Overmyer: Revolutionary vs. Evolutionary Rapid Prototyping: Balancing Software Productivity and HCI Design Concerns. Center of Excellence in Command, Control, Communications and Intelligence (C3I), George Mason University, 4400 University Drive, Fairfax, Virginia.
9. **^** Software Productivity Consortium: Evolutionary Rapid Development. SPC document SPC-97057-CMC, version 01.00.04, June 1997. Herndon, Va. Page 6.
10. **^** Davis. Page 72-73. Citing: E. Bersoff and A. Davis, Impacts of Life Cycle Models of Software Configuration Management. Comm. ACM, Aug. 1991, pp. 104–118
11. **^** Adapted from C. Melissa Mcclendon, Larry Regot, Gerri Akers.

12. **^** Adapted from Software Productivity Consortium. PPS 10-13.
13. **^** Joseph E. Urban: Software Prototyping and Requirements Engineering. Rome Laboratory, Rome, NY.
14. **^** Paul W. Parry. Rapid Software Prototyping. Sheffield Hallam University, Sheffield, UK. [6] (http://www.shu.ac.uk/schools/cms/rapid.software.prototyping/rapid.software.prototyping.html)
15. **^** Dr. Ramon Acosta, Carla Burns, William Rzepka, and James Sidoran. Applying Rapid Prototyping Techniques in the Requirements Engineering Environment. IEEE, 1994. [7] (http://www.stsc.hill.af.mil/crosstalk/1994/oct/xt94d10g.html)
16. **^** Software Productivity Solutions, Incorporated. Advanced Requirements Engineering Workstation (AREW). 1996. [8] (http://www.sps.com/company/techfocus/modeling/arew.html)
17. **^** from GE Research and Development. http://www.crd.ge.com/esl/cgsp/fact_sheet/objorien/index.html
18. **^** Dynamic Systems Development Method Consortium. http://na.dsdm.org
19. **^** Alan Dix, Janet Finlay, Gregory D. Abowd, Russell Beale; Human-Computer Interaction, Third edition

References

1. Smith MF *Software Prototyping: Adoption, Practice and Management*. McGraw-Hill, London (1991).
2. Dewar, Robert B. K.; Fisher Jr., Gerald A.; Schonberg, Edmond; Froelich, Robert; Bryant, Stephen; Goss, Clinton F.; Burke, Michael (November 1980). "The NYU Ada Translator and Interpreter". *ACM SIGPLAN Notices - Proceedings of the ACM-SIGPLAN Symposium on the Ada Programming Language*. **15** (11): 194–201. ISBN 0-89791-030-3. doi:10.1145/948632.948659 (https://doi.org/10.1145%2F948632.948659).
3. SofTech Inc., Waltham, MA (1983-04-11). "Ada Compiler Validation Summary Report: NYU Ada/ED, Version 19.7 V-001" (http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA136759). Retrieved 2010-12-16.
4. [1] (http://garmahis.com/reviews/wireframe-tools/) List of common UI prototyping tools
5. How Simulation Software Can Streamline Application Development (http://www.cio.com/article/print/28501)
6. [2] (http://konigi.com/node/1416) Archived (https://web.archive.org/web/20090919025431/http://konigi.com/node/1416) September 19, 2009, at the Wayback Machine.
7. Top 10 Simulation Tools for UI Designers, Information Architects and Usability Specialists (http://uidesign-usability.blogspot.com/2007/03/top-10-simulation-tools-for-ui.html)
8. Visio Replacement ? You Be the Judge (http://www.boxesandarrows.com/view/visio_replaceme)

9. Luqi; Berzins, Yeh (October 1988). "A Prototyping Language for Real-Time Software". *IEEE Transactions on Software Engineering*. **14** (10): 1409–1423. doi:10.1109/32.6186 (https://doi.org/10.1109%2F32.6186).
10. Luqi; Ketabchi (March 1988). "A Computer-Aided Prototyping System". *IEEE Software*. **5** (2): 66–72. doi:10.1109/52.2013 (https://doi.org/10.1109%2F52.2013).
11. Luqi (May 1989). "Software Evolution through Rapid Prototyping". *IEEE Computer*. **22** (5): 13–25. doi:10.1109/2.27953 (https://doi.org/10.1109%2F2.27953).

Categories: Software development
