Methods for Software Prototyping

Software stakeholders, testers and end-users find it very difficult to express the real requirements. It is almost impossible to predict how a system will affect end product, how the software will interact with other existing systems and what user operations should be automated. Careful requirements analysis along with systematic reviews of the requirements help to reduce the uncertainty about what the system is intended to do and how it will perform. Therefore, there is no real substitute for trying out a requirement before committing to it. Trying out a requirement is possible if a prototype of the system to be developed is established.

A function described in a specification may seem useful and well defined. However, when that function is used with functions, the users often find that their initial concept was incorrect or incomplete. System prototypes allow stakeholders and users to experiment with requirements and to see how the system supports their concepts and ideas. Prototyping is a means of requirements validation. Users discover requirements errors, omissions and specifications early in the software life cycle process.

Software prototyping and hardware prototyping have different objectives. When developing hardware systems, a prototype is normally used to validate the system design. An electronic system prototype may be developed using off-the-shelf components before investment is made in expensive, special-purpose integrated circuits to implement the production version of the system. A software prototype is not normally intended for design validation but to help develop and check the reality of the requirements for the system. The prototype design is usually different from that of the final system.

Six benefits of developing a prototype early in the software life cycle process are:

1. Misunderstandings between software developers and users may be identified as the system functions are demonstrated.

2. Missing user services may be detected.

3. Difficult-to-use or confusing user services may be identified and refined.

4. Software development staff may find incomplete and/or inconsistent requirements as the prototype is developed.

5. A working, but limited, system is available quickly to demonstrate the feasibility and usefulness of the application to management.

6. The prototype serves as a basis for writing the specification for a production quality system.

According to Ince and Hekmatpour the principal purpose of prototyping is to validate software requirements, software prototypes also have other uses (Ince and Hekmatpour, 1987):

1. *User training*: A prototype system can be used for training users before the final system has been delivered.

2. *System testing*: Prototypes can run back-to-back tests. This reduces the need for tedious manual checking of test runs. The same test cases are submitted to both the prototype and the system under test. If both systems give the same result, the test case has not detected a fault. If the results are different, this implies that the tester should look in more detail at the reasons for the difference.

One way to view prototyping is as a technique of risk reduction. A significant risk in software development is requirements errors and omissions. The costs of fixing requirements errors at later stages in the process can be very high. Experiments have shown that prototyping reduces the number of problems with the requirements specification and the overall development costs may be lower if a prototype is developed. (Boehm et at. 1984)

A process model for prototype development is shown in Figure 1. The objectives of prototyping should be made explicit from the start of the process. The objective may be to develop a system to prototype the user interface; it may be to develop a system to validate functional system requirements; or it may be to develop a system to demonstrate the feasibility of the application to management. The same prototype cannot meet all objectives. If objectives are left implicit, stakeholders, management or end-users may misunderstand the function of the prototype. Consequently, they may not get the benefits that they expected from the developed prototype.

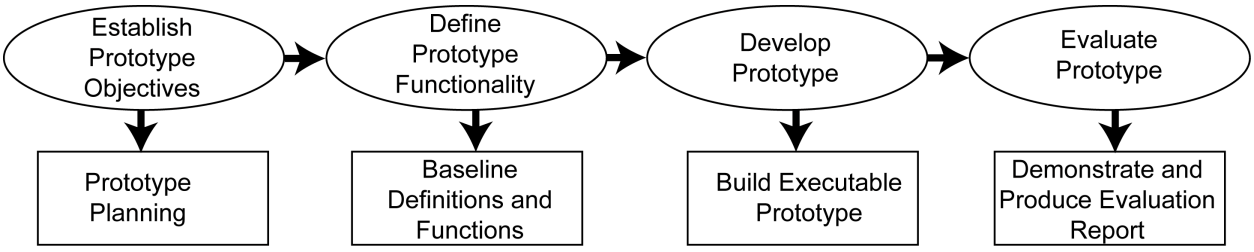

Figure 1. Process Model for Prototype Development

The next stage in the process is deciding what to put into and. perhaps more importantly, what to leave out of the prototype system. Software prototyping is expensive if the prototype is implemented using the same tools and the same standards as the final system. Thus, it may be decided to prototype all system functions but at a reduced level.

Alternatively a subset of system functions may be included in the prototype. Normal practice in prototype development is to relax the non-functional requirements. These may include requirements like response time or memory utilization. Error handling and error management may also be ignored or may be minimal unless the objective of the prototype is to establish a user interface. Standards of reliability and program quality may also be reduced.

The final stage of the process model is evaluating the prototype. Ince and Hekmatpour suggest that this is the most important stage of prototyping. Provision must be made during this stage for testers and user training. Also, the prototype objectives should be used to derive test plans and user scripts for evaluating the prototyped system. Stakeholders, testers and users need time to become comfortable with the new system and to become familiar with the normal modes of usage. Once they are using the system as they expect it to operate, they then discover requirements errors, omissions or additional features they would like added.

In today's environment one major technical problems associated with prototyping involves the need for rapid software development. However, there are non-technical, managerial problems, which may make it difficult to use prototyping in some organizations:

- Planning, costing and estimating a prototyping project is outside the experience of many software project managers.

- Procedures for change and configuration management may be unsuitable for controlling the rapid change inherent in prototyping. However, if there are no change management procedures, evaluation is difficult; the evaluators are trying to assess a constantly changing system.

- Managers may exert pressure on prototype evaluators to reach swift conclusions about the prototype. These may result in inappropriate requirements.

A common argument against prototyping is that the cost of prototyping represents an unacceptably large fraction of the total development costs. It may be more economic to modify a finished system to meet unperceived needs than to provide an opportunity for stakeholders and users to understand and refine their needs before the final system is built.

This may be true for some systems but effective prototyping increases software quality. It can give software developers a competitive edge over their competitors. In the 1960's and 1 970's, the US and European automobile industries did not invest in quality procedures. They lost a great deal of their market share to higher quality Japanese automobiles. This is an illustration of how a "build it cheap and fix it later" philosophy can be extremely expensive in the long term.

Although prototyping is now a well-established approach, there is little reported information about the costs of prototyping and its influence on total system costs. An early study was reported by Gomaa (1983) who describes the advantages of developing a prototype for a process management and information system. Prototype development costs were less than 10% of the total system costs. In the development of the production-quality system, there were no requirements definition problems. The project was completed on time and the system was well received by users. Bernstein (1993) has also reported user satisfaction and a reduction in development time of 40% more recently.

Prototyping is a key technique in the spiral process model for risk evaluation. By developing a prototype, requirements and design risks can be reduced. Short-term additional costs may result in long-term savings as requirements and design decisions are mitigated during the prototyping

process.

Prototyping in the software process

It is difficult for stakeholders and the end-users to anticipate how they will use the new software systems and how it will respond to support their everyday work. If these systems are large and complex, it is probably impossible to make this assessment before the system is built and installed on the target system.

One way of tackling this difficulty is to use an evolutionary or iterative approach to systems development. This means giving the user a system, which is incomplete, and then modifying and augmenting it iteratively, as the user requirements become clearer. Another decision might be made to build a "throwaway" prototype to help requirements analysis, validation, tester and user to establish their needs. After evaluation, the throwaway prototype is discarded and a production-quality system built.

The distinction between these two approaches is that evolutionary prototyping starts out with a limited understanding of the system requirements and the system is augmented and changed as new requirements are discovered. There may never be a system specification document. The software systems developed by the evolutionary approach may be unspecified. This may be that the system and software engineers don't understand or know all the process and ramifications of the system they are trying to automate. For example, it is impossible to specify many types of Artificial Intelligent system because we don't understand how people solve problems. By contrast, the throwaway prototype approach is intended to discover the system specification so that the output of the prototype development phase becomes the specification from which the requirements are derived from.

The important differences between the objectives of evolutionary and throw-away programming is:

- The objective of evolutionary prototyping is to deliver a working system to end-users.
- The objective of throwaway prototyping is to validate or derive the system requirements.

In the evolutionary case, the first priority is to incorporate well-understood requirements in the prototype then move on to the requirements that are unclear. In the throwaway prototype, the priority is to understand requirements that are unclear. The system engineers therefore start with those requirements that are not well understood. Requirements, which are straightforward probably don't need to be prototyped.

Evolutionary prototyping

Evolutionary prototyping is based on the idea of developing an initial implementation, exposing this to user comment and refining this through many stages until an adequate system has been developed Figure 2.

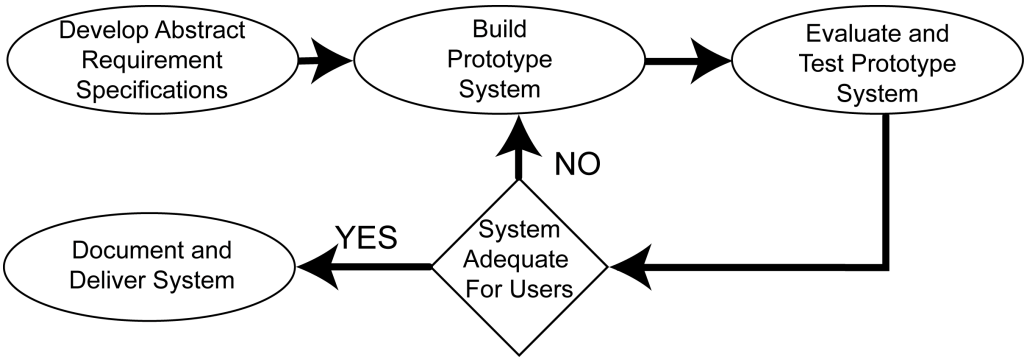

Figure 2. Evolutionary prototyping

Evolutionary prototyping is one realistic way to develop systems where it is difficult or impossible to establish a detailed system specification document. The key to success in the evolutionary prototyping approach is to use techniques, which allow for rapid system iterations. Suggested changes may be incorporated and demonstrated as quickly as possible. This may mean using high level programming language such for software development. Special-purpose environments and integrated software tools may be used to support and accelerate the development process.

An important difference between evolutionary prototyping and a specification-based approach to development is in verification and validation. Verification is only meaningful when a program is compared to its specification. If there is no specification, verification is impossible. The validation process should demonstrate that the program is suitable for its intended purpose rather than its conformance to a specification.

The systems adequacy is not measurable and only a subjective judgment of a program 5 adequacy can be made. This does not invalidate its usefulness; human performance cannot be guaranteed to be correct but we are satisfied if performance is adequate for the task in hand.

There are three problems with evolutionary prototyping, which are particularly important when large, long-lifetime systems are being developed:

1. Existing software management structures are set up to deal with a software process model that generates regular deliverables to assess progress. Prototypes usually evolve so quickly that it is not cost-effective to produce a great deal of system documentation and schedules.

2. Continual change tends to corrupt the structure of the prototype system. Maintenance is therefore likely to be difficult and costly. This is particularly likely when the system maintainers are not the original developers. The development teams are hardly ever responsible for system maintenance.

3. It is not clear how the range of skills, which is normal in software engineering teams, can be used effectively for this mode of development. Small teams of highly skilled and motivated individuals have implemented the systems developed in this way.

These three problems do not mean that evolutionary prototyping should not be used. It allows systems to be developed and delivered rapidly. System development costs are reduced. If the stakeholders and users are involved in the development, the system is likely to be appropriate for their needs. However, organizations that use this approach must accept that the lifetime of the system will be relatively short. As its structure becomes unmaintainable, it must be completely rewritten.

Throwaway prototyping

A software process model based on an initial throwaway prototyping stage is illustrated in Figure 3. The throwaway prototyping approach extends the requirements analysis process with the intention of reducing overall life cycle costs. The principal function of the prototype is to clarify requirements and provide additional information for managers to assess process risks. After evaluation, the prototype is thrown away. It is not used as a basis for further system development.

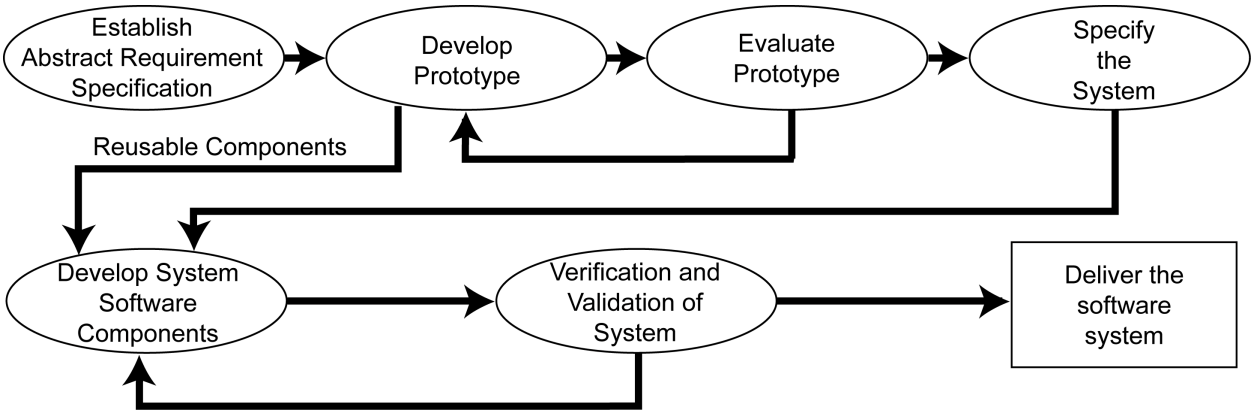

Figure 3 Throwaway prototyping

The process model in Figure 3 assumes that the prototype is developed from an outline or abstract system specification, delivered for experiment and modified until the stakeholder or user is satisfied with its functionality. At this stage, a conventional software process model is entered; a specification is derived from the prototype and the system re-implemented in a final production version. Components from the prototype may be reused in the production-quality system.

The stakeholders and end-users should resist the temptation to turn the throwaway prototype into a delivered system. The reasons for this are:

1. Important system characteristics such as performance, security, robustness and reliability may have been ignored during prototype development so that a rapid implementation could be developed. It may be impossible to tune the prototype to meet these non-functional requirements.

2. During the prototype development, the prototype will have been changed to reflect user needs. It is likely that these changes will have been made in an uncontrolled way. The

only design specification is the prototype code. This is not good enough for long-term maintenance

.

3. The changes made during prototype development will probably have degraded the system structure. The system will be difficult and expensive to maintain.

Rather than derive a specification from the prototype, it is sometimes suggested that the system specification should be the prototype implementation itself. The instruction to the software contractor should simply be "write a system like this one". There are also several problems with this approach:

1. Important features may have been left out of the prototype to simplify rapid implementation. It may not be possible to prototype some of the most important parts of the system such as safety-critical elements.

2. A prototype implementation has no legal standing as a contract between customer and contractor.

3. Non-functional requirements such as those concerning reliability, robustness and safety cannot be adequately tested in a prototype implementation.

A general problem with throwaway prototyping is that the mode of use of the prototype may not correspond with the way that the final delivered system is used. The tester of the prototype may be particularly interested in the system and may not be typical of system users. The training time during prototype evaluation may be insufficient. If the prototype is slow, the evaluators may adjust their way of working and avoid those system features, which have slow response times. When provided with better response in the final system, they may use it in a different way.

Incremental development

An incremental development model combines the advantages of evolutionary prototyping with the control required for large-scale development projects was developed by Mills et al (1980). This incremental development model (Figure 4) involves developing the requirements and delivering the system in an incremental fashion. As a part of the system is delivered, the user may experiment with it and provide feedback to the system developers. Incremental development is a key part of the Cleanroom development process.

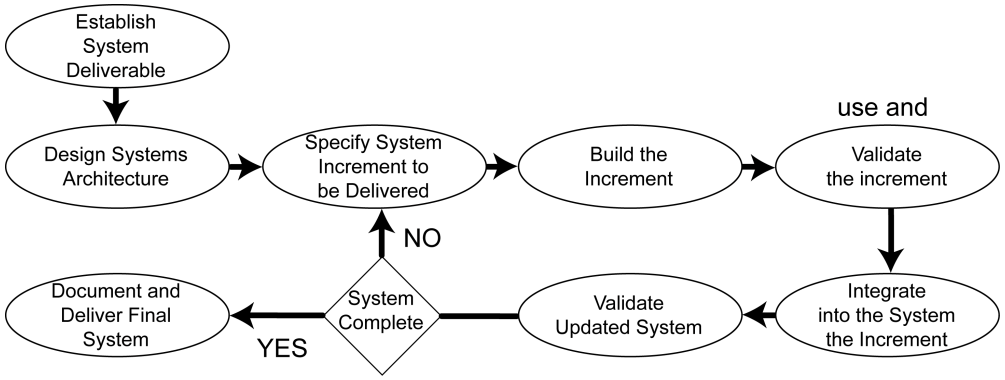

Figure 4. Incremental development

Incremental development avoids the problems of constant change, which characterize evolutionary prototyping. An overall system architecture is established early in the process to act as a framework. System components are incrementally developed and delivered within this framework. Once these have been validated and delivered, neither the framework nor the components are changed unless errors are discovered. User feedback from delivered components however can influence the design of components scheduled for later delivery.

Incremental development is more manageable than evolutionary prototyping as the normal software process standards are followed. Plans and documentation must be produced for each system increment. It allows some user feedback early in the process and limits system errors, as the development team is not concerned with interactions between quite different parts of the software system. Once an increment has been delivered. its interfaces are frozen. Later increments must adapt to these interfaces and can be tested against them.

A problem with incremental development is that the system architecture has to be established before the requirements are complete. This means that the requirements tend to be constrained by the architecture that is established. Another, non-technical problem is ihat this approach to development does not fit well with established contractual models for software developers. Contracts for the development must be flexible and established before the requirements are fixed. Many organizations that use traditional engineering models for software procurement find it impossible to adapt to the form of contract, which this approach requires.

Prototyping techniques

System prototyping techniques should allow the rapid development of a prototype system. As staff costs are the principal software costs, rapid development means that prototype costs are minimized. It also means that feedback from the stakeholder's users can be obtained early in the overall software process.

There are a number of techniques, which have been used for system prototyping. These include:

- Executable specification languages
- Very high-level languages
- Application generators and fourth-generation languages
- Composition of reusable components.

These prototyping techniques are not mutually exclusive. They can be used in combination. One part of the system may be generated using an application generator and linked to reusable components that have been taken from existing systems. Luqi (1992) describes this mixed approach, which was used to create a prototype of a command and control system.

Executable specification languages

If a system specification is expressed in a formal, mathematical language, it may be possible to animate that specification to provide a system prototype. A number of executable formal specification languages have been developed (Lee and Sluizer, *1985;* Henderson and Minkowitz, 1986; Gallimore et al., 1989). DilIer (1994) discusses techniques of animating formal specifications written in Z.

Developing a prototype from a formal specification is attractive in some ways as it combines an unambiguous specification with a prototype. There are no additional costs in prototype development after the specification has been written. However, there are practical difficulties in applying this approach:

1. Graphical user interfaces cannot be prototyped using this technique. Although models of a graphical interface can be formally specified (Took, 1986), these cannot be systematically animated using current windowing systems.

2. Prototype development may not be particularly rapid. Formal specification requires a detailed system analysis and much time may be devoted to the detailed modeling of system functions, which are rejected after prototypes evaluation.

3. The executable system is usually slow and inefficient. Users may get a false impression of the system and compensate for this slowness during evaluation. They may not use the system in the same way they would use a more efficient version. Users may therefore define a different set of requirements from those, which would be suggested if a faster prototype was available.

4. Executable specifications only test functional requirements. In many cases, the non-functional characteristics of the system are particularly important so the value of the prototype is limited.

The developers of functional languages which have been integrated with graphical user interface libraries and which allow rapid program development have addressed some of these problems.

A functional language is a formal language where the system is defined as a mathematical function. Evaluation of that function (which is obviously decomposed into many other functions) is equivalent to executing a procedural program. Miranda (Turner,1985) and ML (Wikstrom, 1988) are practical functional languages, which have been used for the development of non-trivial prototypes.

Functional languages might also be classed as very high-level languages are discussed next. They allow a very concise expression of the problem to be solved. Because of their mathematical basis, a functional program can also be viewed as a formal system specification. However, the execution speed of functional programs on sequential hardware is typically several orders of magnitude slower than conventional programs. This means that they cannot be used for prototyping large software systems.

Very high-level languages

Very high-level languages are programming languages, which include powerful data management facilities. These simplify program development because they reduce many problems of storage allocation and management. The language system includes many facilities, which normally have to be built from more primitive constructs in languages like Pascal, Ada, C++ and Java. Examples of very high-level languages are Lisp (based on list structures), Prolog (based on logic), Smalltalk (based on objects), APL (based on vectors) and SETL (based on sets).

Very high-level dynamic languages are not normally used for large system development because they need a large run-time support system. This run-time support increases the storage needs and reduces the execution speeds of programs written in the language. If performance requirements can be relaxed for the prototype, then the overhead of the runtime support is acceptable.

As well as the application domain, there are other factors, which should influence the choice of prototyping language:

1. The interactive features of the system to be prototyped. Some languages, such as Smalltalk and 4GLs, have better support for user interaction than others.

2. The support environment that is provided with the language. In this respect, Lisp and Smalltalk has far better environments than alternative languages. Outside the business domain, where 4GLs are common, these have been the most widely used prototyping languages.

One of the most powerful prototyping systems for interactive systems is the Smalltalk system (Goldberg and Robson, 1983). Smalltalk is an object-oriented programming language, which is tightly integrated with its environment. It is an excellent prototyping language for three reasons:

1. The language is object-oriented so systems are resilient to change. Rapid modifications of a Smalltalk system are possible without unforeseen effects on the rest of the system. Indeed, Smalltalk is only suitable for this style of development.

2. The Smalltalk system and environment is an inherent part of the language. All the objects defined in the environment are available to the Smalltalk programmer. Thus, a large number of reusable components are available that may be incorporated in the prototype under development.

3. Some versions of the language are now packaged with a support system (Visualworks) which partially automates the construction of user interfaces for interactive systems. This is a screen-drawing package for graphical interfaces such as those supported by user interface management systems.

A class of wide-spectrum programming languages, proposed as prototyping languages are REFINE (Smith et al., *1985),* EPROL (Hekmatpour, 1988), and LOOPS (Stefik et al.,

1986).

A wide-spectrum language is a programming language, which combines a number of paradigms. Most languages are based on a single paradigm. Pascal is an imperative language, Lisp is based on functions and lists, Prolog is based on facts and logic and so on. By contrast, a wide-spectrum language may include objects, logic programming and imperative constructs. However, the practical problems of developing efficient implementations of wide-spectrum languages have meant that few commercial language products are available. LOOPS is the only language in this category that is widely used.

As an alternative to using a wide-spectrum language, is the use of a mixed-language approach to prototype development. Different parts of the system may be programmed in different languages and a communication framework established between the parts. Zave (1989) describes this approach to development in the prototyping of a telephone network system. Four different languages were used: Prolog for database prototyping, Awk (Aho et at., 1988) for billing, CSP (Hoare, 1985) for protocol specification and PAISley (Zave and Schell, 1986) for performance simulation.

There is probably no ideal language for prototyping large systems. This is because; different parts of the system are diverse. The advantage of a mixed-language approach is that the most appropriate language for a logical part of the application can be chosen, which speed up a prototype development. The disadvantage is that it may be difficult to establish an interface, which will allow multiple languages to communicate.

Fourth-generation languages

Evolutionary prototyping is now commonly used for developing business systems. These rely on the use of fourth-generation languages (4GLs) for system development. There are many 4GLs and their use usually reduces the lifecycle time for system development.

Fourth-generation languages are successful because there is a great deal of commonality across data processing applications. In essence, these applications are concerned with updating a database and producing reports from the information held in the database. Standard forms are used for input and output.

At their simplest, 4GLs are database query languages such as SQL (Date and Darwen ~ 1993). 4GLs may also package a report generator and a screen form design package with the query language to provide a powerful interactive facility for application generation (Figure 5). Some spreadsheet-type facilities may also be included. 4GLs rely on software reuse where common abstractions have been identified and parameterized. Routines to access a database and produce reports are provided. The programmer need only describe how these routines are tailored and controlled.

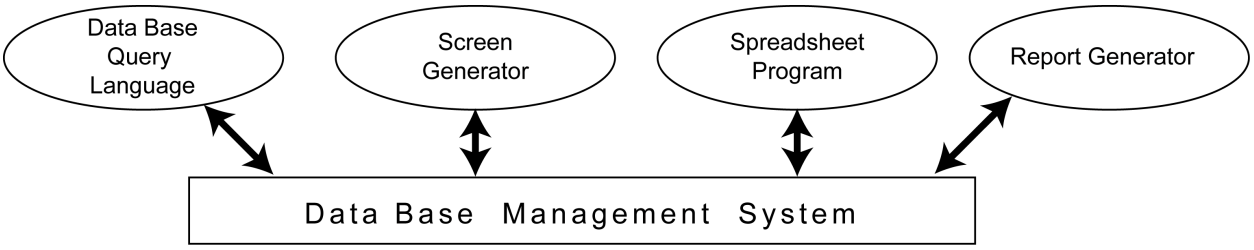

Figure 5

Fourth-generation languages are generally used, often in conjunction with CASE tools, for the development of small to medium-sized systems. The end-users may be involved in the development or may even act as developers. This can result in systems, which are poorly structured and difficult to change.

The vendors of 4GL products claim that system maintenance is simpler because application development time is rapid. The developed applications are usually much smaller than the equivalent COBOL programs. Rather than worry about structuring the system for maintenance, requirements changes are implemented by a complete system rewrite.

Using 4GLs for developing data processing systems is cost-effective in some cases, particularly for relatively small systems. However, 4GLs are slower than conventional programming languages and require much more memory. In an experiment in which Sommerville (1996) was involved, rewriting a 4GL program in C++ resulted in a 50% reduction in memory requirements. The program also ran 10 times faster than before.

There is no standardization or uniformity across fourth-generation languages. This means that users may incur future costs of rewriting programs because the language in which they were originally written is obsolete. Although they clearly reduce systems development costs, the effect of 4GLs on overall life cycle costs for large DP systems is not known. They are obviously to be recommended for prototyping but the lack of standardization may mean that total life cycle cost savings may be less than anticipated.

Some CASE toolsets are closely integrated with 4GLs. Using such systems has the advantage that documentation is produced at the same time as the prototype system. The generated system should be more structured and easier to maintain. These tools may generate 4GL code or may generate code in a lower-level language such as COBOL. Forte (1992) describes a number of tools of this type in a brief survey of fourth-generation languages.

4GL-based development can be used either for evolutionary prototyping or may be used in conjunction with a method-based analysis where system models are used to generate the prototype system. The structure that CASE tools impose on the application means that evolutionary prototypes developed using this approach should be maintainable.

Composition of reusable components

The time needed to develop a system can be reduced if many parts of that system can be reused rather than designed and implemented. Prototypes can be constructed quickly if you have a library of reusable components and some mechanism to compose these components into systems. The composition mechanism must include control facilities and a way of interconnecting components. This approach is illustrated in Figure 6.

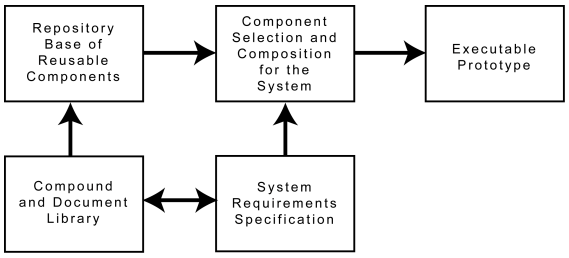

Figure 6, Composition of reusable components

Prototyping with reusable components involves developing a system specification by taking account of what reusable components are available. These components are taken from a repository and put together to form the prototype system. This approach is usually most suitable for throwaway prototyping as the specification may not be exactly what is required. The prototype demonstrates what is possible. The reusable components may be used in the final system to reducing the development cost.

Perhaps the best example of this approach to prototyping is found in the Unix operating system. The features of Unix, which make it particularly suitable for prototyping with reusable components, include:

- Various shell programming languages (Rosenberg. 1991), which may be used as the composition mechanism for reusable components. Unix shells are command languages, which include looping, and decision constructs. They provide facilities for combining commands, which operate on files, integers and character strings.

- A set of functions that have been designed so that they can be combined in various ways. Functions usually rely on simple character stream interfaces, which mean that they are easy to connect. Examples of these functions are *grep* (a paffern matcher), *sort* (a sorting program), and *we* (a word counter).

- Its command interconnection mechanism (pipes) combined with its model of files and I/O devices as character streams. This makes it easy to connect functions to each other and to files and peripherals.

However, prototyping-using Unix is limited because the granularity of the software components is relatively coarse. The function of the individual components is often too general-purpose to combine effectively with other components. Furthermore user interface prototyping using the shell is limited because of the simple I/O model adopted by the Unix system.

Prototyping using reusable components is often combined with other approaches using very high level or fourth-generation languages. The success of Smalltalk and Lisp as prototyping languages is due to their reusable component libraries and to their built-in language facilities.

User interface prototyping

Graphical or forms-based user interfaces have now become the norm for interactive systems. The effort involved in specifying, designing and implementing a user interface represents a very significant part of application development costs. The user must take part in the interface design process. This realization has led to an approach to a design called user-centered design (Norman and Draper, 1986), which depends on interface prototyping and user involvement throughout the interface design stage.

Design means the "look and feel" of the user interface. Evolutionary prototyping is used in the process. An initial interface is produced, evaluated with users and revised until the user is satisfied with the system. After an acceptable interface has been agreed on, it then may be re-implemented, although if interface generators are used this may not be necessary. Interface generators allow interfaces to be specified and a well-structured program is generated from that specification. Thus the iterations inherent in exploratory programming do not degrade the software structure and re-implementation is not required.

Interface generation systems may be based around user interface management systems (Myers, 1988) which provide basic user interface functionality such as menu selection, object display and so on. They are placed between the application and the user interface and provide facilities for screen definition and dialogue specification. These facilities may be based on state transition diagrams for command specification (Jacob, 1986) or on formal grammars for dialogue design (Browne, 1986). A survey of tools for user interface design is given by Myers (1989).

Very high-level languages like Smalltalk and Lisp have many user interface components as part of the system. These can often be modified to develop the particular application interface required. Fourth-generation language systems usually include screen definition facilities whereby picking and positioning form fields can define screen templates.

From a software engineering point of view, it is important to realize that user interface prototyping is an essential part of the process. Unlike the prototyping of system functionality, it is usually acceptable to present an interface prototype system specification. Because of the dynamic nature of user interfaces, paper specifications are not good enough for expressing the user interface requirements.

References

Aho, A. V., Kemighan, B. W. and Weinberger, P. J. (1988). The Awk Programming Language. Englewood Cliffs NJ: Prentice-Hall [149, 710]

Bemstein, L. (1993). Get the design right! IEEE Software, 10 *(5),* 61-3 [140]

Boehm, B. W., Gray, T. E. and Seewaldt, T. (1984). Prototyping versus specifying a multi-project experiment. IEEE Transactions on Software Engineering, SE. 10 (3), 290-303 [139]

Browne, D. P. (1986). The formal specification of adaptive user interfaces using command language grammar. In Proc. CHI'86, Boston, 256-60 [152]

Date, C. J. and Darwen, H. (1993). A Guide to the SQL Standard, 3rd edn. Reading MA: Addison-Wesley [534,708]

Diller, A. (1994). Z: An Introduction to Formal Methods, 2nd ed. New York: John Wiley and Sons [146,190]

Forte, G. (1992). Tools fair: out of the lab, onto the shelf. IEEE Software, 9 (3), 70-9 *[150]*

Gallimore, R. M., Coleman, D. and Stavridou, V. (1989). UMIST OBJ: a language for executable program specifications. Comp. J., 32 (5),413-21 [146]

Goldberg, A. and Robson, D. (1983). Smalltalk-80. The Language and its Implementation. Reading MA: Addison-Wesley [148, 215]

Gomaa, H. (1983). The impact of rapid prototyping on specifying user requirements. ACM Software Engineering Notes, 8 (2), 17-28 [140]

Hekmatpour, S. and Ince, D. (1988). Software Prototyping, Formal Methods and VDM. Wokingham: Addison-Wesley [148, 163]

Hoare, C. A. R. (1985). Communicating Sequential Processes. London: Prentice-Mall [149, 168]

Jacob, R. (1986). A specification language for direct-manipulation user interfaces. ACM Trans. on Graphics, 5 (4), 318-44 [152]

Lee, S. and Sluizer, 5. *(1985).* On using executable specifications for high-level prototyping. In Proc. 3rd mt. Workshop on Software Specification and Design, 130-4 [146]

Luqi (1992). Computer-aided prototyping for a command and control system using CAPS. IEEE Software. 9 (1), 56-67 [146]

Mills, H. D., O[1]Neill, D., Linger, R. C., Dyer, M., and Quinnan, R. E. (1980). The management of software engineering. IBM Sys. J., 24 (2),414-77 [144]

Myers, B. (1988). Creating User Interfaces by Demonstration. New York: Academic Press *[152,558]*

Myers, B. (1989). User-interface tools: introduction and survey. IEEE Software, 6 (1), 15-23 *[152]*

Norman, D. A. and Draper, S. W. (1986). User-centered System Design. Hillsdale NJ: Lawrence Erlbaum [152, 323]

Smith, D. R., Kotik, G. B. and Westfold, S. J. (1985). Research on knowledge-based software environments at Kestrel Institute. IEEE Trans. on Software Engineering, SE-ll (11), 1278-95 [148]

Stefik, M. J., Hobrow, D. G. and Kahn, K. M. (1986). Integrating access-oriented programming into a multiparadigm environment. IEEE Software, 3 (1), 10-18 ~ 148]

Took, R. (1986). The presenter-a formal design for an autonomous display manager. In Software Engineering Environments (Sommerville, I., ed.). Stevenage: Peter Perigrinus, 151-69 [146]

Turner, D. A. *(1985).* MIRANDA: A non-strict functional language with polymorphic types. Proc. Conf. on Functional programming and Computer Architecture, Nancy, France, 1-16 [147]

Wikstrom, A. (1988). Standard ML Englewood Cliffs NJ: Prentice-Hall [147]

Zave, P. (1989). A compositional approach to multiparadigm programming. IEEE Software. 6 (5); 15-27 [123, 148]

Zave, P. and Schell, W. (1986). Salient features of an executable specification language and its environment. IEEE Trans. on Software Engineering, SE- 12 (2),3 12-25 [149]