

7

Mixed Approaches*

Lars Mathiassen
Thomas Seewaldt
Jan Stage

Abstract. Which approaches to software development should be used in which situations? This fundamental question is explored based on experiences from nine comparable small-scale software projects using prototyping and specifying.

An empirical interpretation suggests that mixed approaches to software development will benefit from the strengths of both specifying and prototyping. On functionality, robustness, ease of use, and ease of learning mixed approaches led to products of a quality that was at least comparable to the products of specialized approaches based on either specifying or prototyping. Moreover the Spiral Model was experienced as a useful framework for combining specifying and prototyping approaches to software development.

A theoretical interpretation relates these practical lessons to The Principle of Limited Reduction. This principle suggests that effective software development must cope with both complexity and uncertainty. This requires a systematic effort combining analytical and experimental approaches, independently of whether specifications or prototypes are used.

Keywords: prototyping, specifying, spiral model, complexity, uncertainty.

1. Introduction

Software design situations are characterized by complexity and uncertainty. The degree of complexity represents the amount of relevant information that is available in a given situation as a basis for making design decisions. In contrast, the degree of uncertainty represents the availability and reliability of information that could be relevant for the same purpose (Mathiassen *et al.* 1990, 1992). Software developers are advised to cope with design *complexity* through abstraction and decomposition (DeMarco 1979; Dijkstra 1972; Langefors 1966; Wirth 1973; Wulf 1977; Yourdon 1982). This approach is often referred to as specifying because it involves extensive use of specifications. Software developers are also concerned with *uncertainty*, and new experimental approaches based on prototypes have emerged as means to cope with software design uncertainty (Boehm 1988; Budde *et al.* 1992; Davis 1982; Floyd 1984; Gomaa *et al.* 1981; Gould *et al.* 1985).

In this article, we discuss the use of specifying and prototyping approaches from both a practical and a theoretical point of view. Empirically, our discussion is based on two software engineering experiments in which nine comparable small-scale projects used different approaches to develop the same application software product. In addition, the practical lessons are interpreted in the light of a general theory of software development called The Principle of Limited Reduction, cf. Mathiassen *et al.* (1990, 1992). This principle suggests that effective software development must cope with both complexity and uncertainty. This requires a systematic effort combining analytical and experimental approaches, independently of whether specifications or prototypes are used. The conclusion, which is consistent with the practical lessons as well as the theoretical interpretation, is that software engineers should rely on mixed approaches to software design.

The first small-scale experiment, which has been reported earlier in Boehm *et al.* (1984), was conducted at UCLA in 1982. Different development teams used either a specifying or prototyping approach in the whole development process. Below, we refer to this as the UCLA experiment. The results of this experiment were used to compare the relative strengths and weaknesses of the two specialized approaches. As a consequence of their findings, the authors suggest that for most large projects, and many small ones, a mix of prototyping and specifying is preferable to the exclusive use of ei-

ther approach by itself. In particular, they suggest to use risk management as an effective means to design and manage the specific mix of prototyping and specifying that is necessary to cope with the complexity and uncertainty of a given design situation.

Most contemporary writers agree with the main conclusion of the UCLA experiment. It is generally suggested that different situations in software development calls for different approaches involving both specifications and prototypes (Andersen *et al.* 1990; Boehm *et al.* 1984; Davis 1982; Floyd 1987; Stage 1989). Yet there are only few authors who provide a systematic framework for selecting and mixing these approaches. Some authors claim that a combination is possible and they mention a number of advantages and disadvantages of each approach. They do, however, give very little advice on how to actually combine them, e.g. Fairley (1985) or Pressman (1987).

Boehm's (1988) Spiral Model handles the mixing of specifying and prototyping more systematically. Each cycle of the spiral includes a re-evaluation of risks and subsequent development of prototypes and specifications, cf. figure 1. The main emphasis is on resolving sources of risk. In effect, the Spiral Model is a constructive attempt to dynamically combine and mix specialized approaches during a development effort.

The second small-scale experiment, reported in this article, replicates, as far as possible, the research method and results of the UCLA experiment. Two development teams used a mixed approach, based on the Spiral Model, to develop the same application software product. The experiment was carried out at Aalborg University in 1990. Below, it is referred to as the AU experiment. It was designed to investigate the following questions:

- How does a mixed approach to software development compare to specialized approaches based on either specifications or prototypes?
- Does the Spiral Model provide a useful framework for managing software projects based on a mixture of specifications and prototypes?

Section 2 of this article summarizes the UCLA projects, involving seven teams, and it describes, in detail, the AU projects, involving two teams. Section 3 surveys the major experiences of the AU projects, compares them to the experiences of the UCLA projects, and

presents a supplementary interpretation of the use of the Spiral Model. Finally, section 4 provides a theoretical interpretation that relates the findings of the projects to The Principle of Limited Reduction.

2. The experiments

The AU experiment involved two student teams developing the same product as in the UCLA experiment. These software projects constituted the practical part of the students' second graduate semester study project (Larsen *et al.* 1990). Such study projects account for half of the time in that semester and half of the major subject grade. Two of the authors, L. Mathiassen and J. Stage, were supervisors.

This section describes the key aspects of the AU software projects: the product developed, the Spiral model that was applied in the development of the product, the development environment, the organization and staffing of the two teams, the experimental data collection procedures, and the main similarities and differences between this experiment and the UCLA experiment. To enable comparison, we provide a brief summary of the results from the UCLA experiment.

2.1. The UCLA Projects

In the UCLA experiment (Boehm *et al.* 1984), seven teams of students developed the same application software product. The product was an interactive computer system supporting the COCOMO software cost estimation model (Boehm 1981). The systems and the related documentation were developed over 10 weeks and comprised roughly 3000 lines of Pascal source instructions.

All teams were required to collect data on their efforts and products during the whole experiment. Four out of the seven teams applied a specifying approach. They produced a requirements specification, a design specification, and a final product that included operational code, user manual, and maintenance manual. The remaining three groups applied a prototyping approach. They produced the same final product, but they were required to produce and exercise a prototype by the midpoint of the 10 weeks for which the experiment lasted.

The main conclusion was that each approach focuses only on some of the properties that characterize software of high quality. The specifying approaches produced more coherent designs and software that was easier to integrate. The products were rated higher on functionality and robustness and lower on ease of learning and ease of use. The prototyping approaches yielded products with roughly the same performance, but with about 40 percent less code and 45 percent less effort. The products were rated higher on ease of learning and ease of use and somewhat lower on functionality and robustness. For this reason, the authors concluded that the specifying and prototyping approaches seem to complement each other.

2.2. The AU Projects

The AU experiment involved two student teams. Each team should develop the same application software product as in the UCLA experiment, i.e., an interactive computer system supporting the COCOMO software development cost estimation model. In this model, calculations are based on computer system components being described tentatively in terms of size and ratings with respect to 16 specific cost-driver attributes, e.g. hardware constraints, database size, personnel skills and experience, and use of tools and modern programming practices. These attributes are used to calculate the amount of time and effort required to develop each of the components as well as the overall system.

The algorithms and tables of the model were provided in Boehm (1981) but each team was to design its own user interface and file system. The user interface of this product is considerably more extensive than the calculation algorithm. It must support selective creation, modification, and deletion of the cost-driver parameters describing each component of a software product. It must support the generation and formatting of selected output including overall cost, effort, schedule estimates, and their breakdown by component, phase, and activity. Finally, it must detect and provide messages for erroneous input and provide some level of on-line help facilities.

The design of the interface involves decisions on a further variety of options and alternatives including the use of menus, commands, tables, and forms for input as well as the selection of different output.

Finally, the teams were encouraged to develop a graphical user interface. The main reason for this difference to the UCLA experiment, where line oriented terminals were used, was that it seemed unnecessary to ignore the technical options that had become available since 1982.

2.3. The Spiral Model

The purpose of the AU experiment was to investigate software design based on both specifying and prototyping approaches. This mixing of approaches may be achieved in several, different ways. In both projects, development was based on Boehm's (1988) Spiral Model. This model introduces a systematic way of mixing specifying and prototyping approaches. The choice of either of these ap-

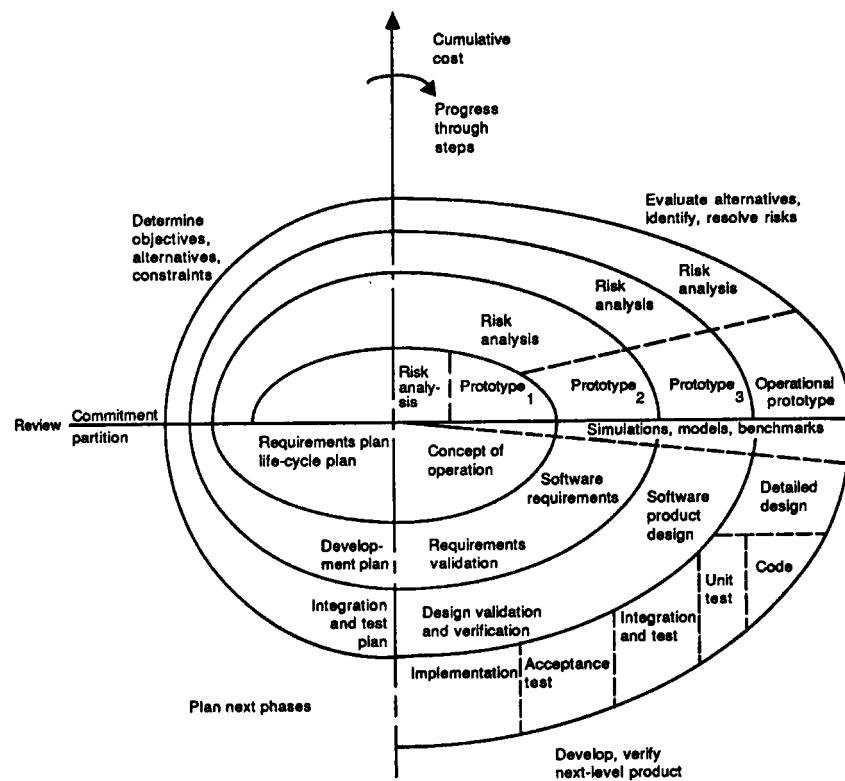


Figure 1. The Spiral Model of software development, from Boehm (1988).

proaches is based on an analysis of the risk factors that are significant to the development project considered. With the Spiral Model, software development is generally divided into a number of cycles where each cycle involves a progression and comprises the same types and sequence of activities. Taken together, the cycles comprise a spiral movement as shown in figure 1. In the figure, the radial dimension represents the cumulative costs of the activities carried out. The angular dimension represents the progress made in each cycle of the spiral.

A typical cycle starts with determination of the outcome of this cycle. This involves objectives, alternatives, and constraints of the products being elaborated in this cycle. The key issue in the next step of the cycle is to identify uncertainties that contribute significantly to project risk. This is done through evaluation of alternatives relative to objectives and constraints. Finally, this step includes formulation of a strategy for resolving the main sources of risks. The third step comprises development and verification of the product of the cycle. If the risk is high, some effort is made to resolve the sources of uncertainty. This may involve specifying as well as prototyping. When all the main sources of risk have been resolved, development follows the waterfall model. The purpose of the fourth step is to develop plans for the next cycle. This may include division of the product into components to be developed separately. Finally, the transformation from one cycle to the next is based on a review of the products of the present cycle and the plans for the next cycle.

2.4. The development environment

The final products were mainly programmed in Modula-2 on Sun-3 work stations under the Unix operating system. Compared to the version of Pascal used in the UCLA experiment, the main advantages of Modula-2 are better facilities for input/output, string handling, and separate compilation with strong type checking. During development no debugger to Modula-2 was available.

The graphical user interfaces were programmed in a graphical tool chosen independently by the two teams. One team used Suntools and the other team used Tooltool.

Finally, the teams had the opportunity to use Hypercard for the Macintosh for development of early prototypes. Only one of the

two teams used this environment. The other team used Tooltool for the early prototypes.

2.5. Team organization and staffing

During the design of the experiment, the 5 participating students were divided into two teams according to their own choice. Below, the two teams are referred to as SM1 and SM2.

Both teams were given the opportunity to organize their work in whatever way they found appropriate. The team with three members (SM1) conducted formal meetings and the rest of the time they worked as separately as possible.

The team with two members (SM2) worked more closely together except when developing different prototypes and during programming and implementation.

2.6. Acceptance test

In the UCLA projects, the performance rating was made by the authors of (Boehm *et al.* 1984). Firstly, they exercised each student team's product together to check its performance. Secondly, they independently exercised each product in more detail and rated it on a scale of 0 to 10 with respect to the following four performance criteria:

1. *Functionality*: The functional capability of the product, i.e., the relative utility of the various computational, user interface, output, and file management functions.
2. *Robustness*: The degree of protection from aborts, crashes, and loss of data provided by the product.
3. *Ease of Use*: The degree of user convenience when performing desired functions, the degree of conceptual clearness and coherence in the user interface, and the avoidance of over-constrained or unexpected program behavior. Boehm *et al.* (1984) also characterize this property as lack of frustration when using the product.
4. *Ease of Learning*: The ease with which new users can master the product's workings and make it do what they wish. The rating on this property also included an evaluation of the user manual and other kinds of documentation supporting the use of the product.

The acceptance tests of the AU projects were carried out by all three authors of this article. L. Mathiassen and J. Stage exercised the products together and discussed their capabilities. Afterwards, they exercised the products independently and rated them with respect to the four performance criteria used in the UCLA experiment. T. Seewaldt exercised and rated the products independently.

2.7. Limitations

The AU projects were designed to resemble the UCLA projects as much as possible. T. Seewaldt participated both in the design of the AU experiment and in the evaluation of the systems that were developed. He also took part in the original UCLA experiment as well as in the reporting of it.

The detailed description of the UCLA projects, cf. (Boehm *et al.* 1984), and the involvement of T. Seewaldt facilitated a comparable design of the AU projects. In both experiments, the task was to develop a product with exactly the same functionality, the organization and staffing of teams was almost the same, the participants had comparable programming experience, cf. table 1, and both experiments applied the same rating procedure and performance criteria in the acceptance test. Besides, both experiments involved use of specifications and prototypes.

	UCLA		AU
	<i>Specifying teams</i>	<i>Prototyping teams</i>	<i>Spiral Model teams</i>
Programming	36	53	26
Pascal/Modula-2	7	18	11
Unix	5	3	11

Table 1. Average programming experience (in months) of the teams in the UCLA and AU experiments.

Both experiments involved a small number of teams. The UCLA experiment involved four and three teams, respectively, and the AU experiment involved two. The empirical findings are, for this reason, not conclusive; they should be seen as systematic interpretations of the results and experiences from a small number of comparable projects. In addition, the specific setting of the AU experiment introduced some experimental limitations. These limitations must

be taken into account in comparing the UCLA and the AU experiments and in evaluating the general relevance of the results. The main differences between the UCLA and AU experiments can be summarized in the following way:

Development Environment: The programming languages, programming environment, and computing resources were more powerful in the AU experiment. One should expect this to increase the productivity in the AU experiment. On the other hand, the new and richer possibilities for designing graphical interfaces introduced more complexity which should decrease productivity in the AU experiment.

Development Approach: In the UCLA experiment each team had a specific procedure for performing their project. These procedures were based on either a specifying or a prototyping approach. In the AU experiment, the teams had the general description of the Spiral Model (Boehm 1988), and a definite deadline for delivery of the final product. There were no intermediate deadlines and no pre-defined procedure. Combined with the nature of the Spiral Model itself, this required more management and communication activities.

In the AU experiment, the Spiral Model was used in a disciplined but informal way, not enforcing formal reviews and other kinds of external control or interaction. Each group worked as an autonomous unit, conducting only informal reviews when they found it necessary. In addition, the Spiral Model is open to individual interpretation and the two groups turned out to interpret the model differently.

Motivation and Stress: The AU students had only moderate course activity in parallel with the experiment, whereas the UCLA students took two or three other courses in parallel with their participation in the experiment. The AU students were also involved in designing and evaluating the experiment, whereas the UCLA students did the experiment as part of a pre-destined course without participating in the evaluation of the experiment. Consequently, one should expect the AU students to be less stressed and more motivated than the UCLA students.

Data Collection Procedures: In both experiments, product size was measured in terms of delivered source instructions (DSI), and it was determined by means of the definition in (Boehm 1981). Delivered source instructions include all lines of program code created by the project team. A source instruction is one line of program code

except that comments, blanks, and unmodified utility software components are excluded. If more than one statement or only part of a statement is placed on the same line, it still counts as one source instruction.

The collection of data on development effort was in both experiments based on the same pre-defined categories of activity. Yet some uncertainty is introduced due to individual interpretations of these categories by each person and team. The AU students' high interest in the experiment itself may have resulted in a more accurate collection of experimental data.

The above differences between the UCLA and AU projects make it difficult to compare the *absolute* values of the development effort, the size of the products, and their performance ratings. On the other hand, the distributions of relative effort and product performance on the four criteria allow some interesting conclusions, as we shall see below.

3. Practical results

A comparison of the AU and UCLA projects illustrates some of the relative advantages and disadvantages of mixed versus specialized approaches to software development. Below, we only discuss points on which we find a comparison possible and relevant. On other points, interesting experiences have been obtained. Some of these are briefly mentioned at the end of this section.

3.1. Product size and development effort

The product size and development effort are illustrated in figure 2 and table 2. In the AU experiment, the two teams developed final products with a size of 4114 and 5457 delivered source instructions (DSI). Most of the code was in Modula-2 and the rest in the language of the graphical tool applied for the user interface.

It is difficult to compare the *absolute* size and effort with the results of the UCLA experiment because of the differences mentioned in section 2.7. Some of these differences could be expected to increase and other to decrease productivity. But a comparison of the ratio is possible and relevant. The productivity of the two teams in the AU experiment were 4.6 and 9.3 DSI/ hour with an average on 6.4 DSI/hour. In the UCLA experiment the average productivity of the specifying teams was 5.8 DSI/hour and the prototyping teams

6.3 DSI/hour. These figures suggest that despite the differences in development approach and technology, and the individual variations between teams a simple measure of lines produced/effort gives similar results in all three types of projects.

		UCLA		AU
		Specifying teams	Prototyping teams	Spiral Model teams
Characteristics	Program Size	3391	2064	4786
	Man hours	584	325	743
Productivity	Overall	5.8	6.3	6.4

Table 2. Average product size and development effort in the UCLA and AU experiment teams.

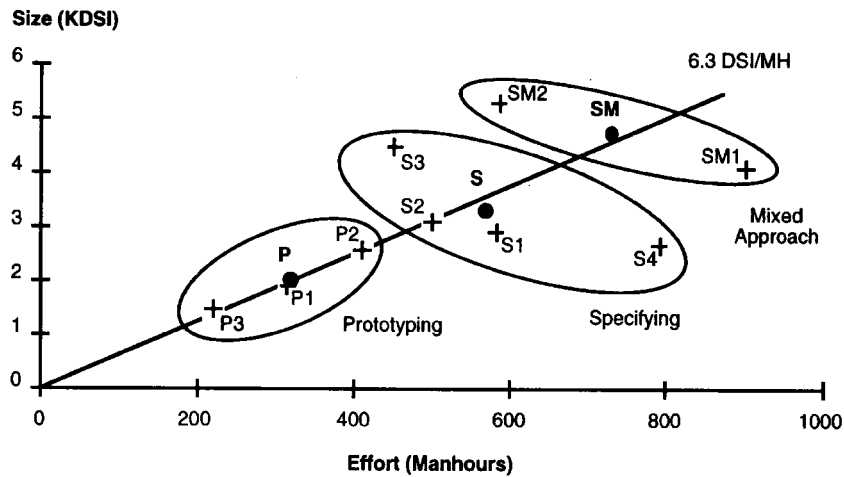


Figure 2. Product size and development effort.

3.2. Distribution of effort by phase

Again, the absolute amount of total effort cannot be compared due to the differences discussed in section 2.7. But the distribution of effort on phases or categories of activity is comparable.

The distribution of effort in the Spiral Model teams combines essential characteristics of the effort distribution of the specifying

MIXED APPROACHES

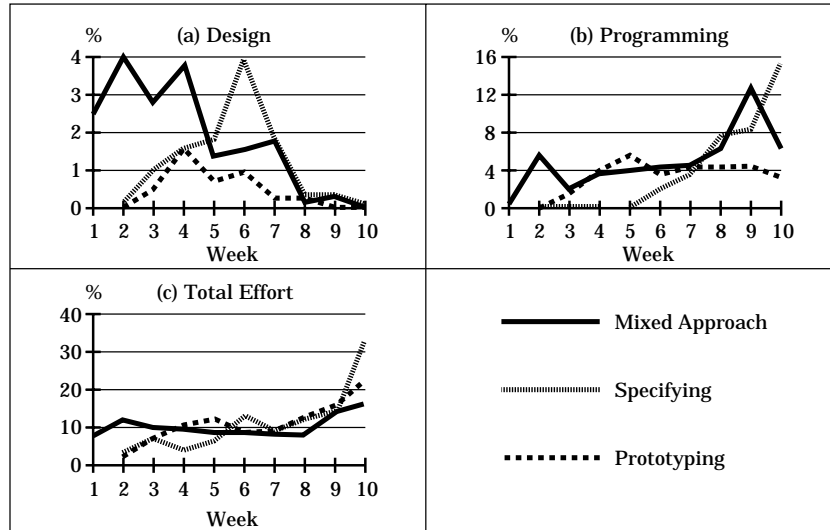


Figure 3. Distribution of effort by phase.

and prototyping teams, cf. figure 3. In the early phases of development, they have a design peak that is similar to the specifying teams, cf. figure 3(a). Due to prototype development, they also have early programming peaks that are similar to the prototyping teams, cf. figure 3(b). This indicates that the Spiral Model leads to a real combination of activities from both the specifying and prototyping approaches.

Furthermore, the Spiral Model seems to support early commencement of design and prototype development, cf. figure 3(a) and (b), and a more even distribution of effort by phase in the whole duration of the project, cf. figure 3(c).

3.3. Product quality

The performance rating of the products in the AU experiment was carried out in the same way as in the UCLA experiment. A comparison of these ratings leads to the following conclusions:

1. The specifying approaches seem to emphasize functionality and robustness at the expense of ease of use and ease of learning.

2. The prototyping approaches and the mixed approaches seem to be more equally concerned with all four performance criteria.
3. The mixed approaches seem to strongly emphasize robustness, thereby avoiding the weakest performance aspect of the prototyping approaches.

In the UCLA experiment, cf. (Boehm *et al.* 1984) and table 3, the specified products have a higher average performance score on functionality and robustness than the prototyped products. On ease of use and ease of learning, the prototyped products rated higher than the specified products.

	UCLA	
	<i>Specifying teams</i>	<i>Prototyping teams</i>
Functionality	6.08	4.78
Robustness	5.13	3.89
Ease of Use	3.25	4.67
Ease of Learning	3.71	4.89
Sum	18.17	18.23

Table 3. Average performance scores in the UCLA experiment.

	AU		
	<i>Spiral Model teams</i>		
	<i>SM1</i>	<i>SM2</i>	<i>Average</i>
Functionality	6.00	8.33	7.17
Robustness	8.00	7.67	7.84
Ease of Use	6.67	7.33	7.00
Ease of Learning	6.33	5.67	6.00
Sum	27.00	29.00	28.00

Table 4. Average performance scores in the AU experiment.

Comparing the AU and UCLA projects, cf. table 3 and table 4, the mixed approaches were at least rated at the same level as the two specialized approaches. Furthermore, the mixed approaches have quite an even distribution of scores on the four performance criteria.

The four aspects considered in the performance rating all stress external properties of the product. They represent the users' point of view. An important question is whether the products of the mixed approaches had the same technical quality as the products of the specialized approaches. In the UCLA experiment this question was handled in the following way: each student was asked to rate each of the other teams' products in the order in which they would prefer to maintain the product.

In the AU experiment, this procedure was impossible as there were only two teams. Instead, both products were evaluated by T. Seewaldt. His conclusion was that the maintainability of the products were comparable between the two experiments. In the AU experiment, the maintenance manuals were far better but the programs contained fewer comments than the programs developed in the UCLA experiment.

3.4. Distribution of code

In the UCLA experiment, each group built a user interface from scratch using a character-oriented terminal as the primary input/output device. Contrary to this, the teams in the AU experiment developed their systems for a graphic work station and they used a powerful graphic library. Due to this, one should anticipate the relative amount of code devoted to the user interface would be lower compared to the UCLA experiment. This expectation did not hold. Despite the technological differences, the distribution of code by function is roughly the same in the UCLA and AU experiments, cf. table 5.

	<i>UCLA</i>		<i>AU</i>
	<i>Specifying</i>	<i>Prototyping</i>	<i>Spiral Model</i>
User Interface	67%	74%	75%
Model Computations	7%	5%	6%
File Management	12%	10%	9%
Miscellaneous	13%	10%	9%

Table 5. Relative distribution of code by function.

These figures indicate that the distribution of code by function depends primarily on the characteristics of the product: a small cost estimation system with an extensive user interface. The distribution

of code by function seems to be independent of the development approach and the applied technology.

3.5. Experiences with the spiral model

The AU projects were designed to make a comparison with the UCLA projects possible, but also to learn about the strengths and weaknesses of the Spiral Model as a practical framework for software development. For the latter purpose, each team of the AU experiment wrote a diary, cf. (Jepsen *et al.* 1989), to record their experiences and evaluations during the project. On the basis of these diaries and the other recordings made, the following lessons were learned about the Spiral Model, cf. (Larsen *et al.* 1990):

- The Spiral Model must be interpreted and adapted to the specific conditions of a project.
- The Spiral Model encouraged the teams to emphasize design considerations before developing the final system.
- The Spiral Model supported the teams in developing code to learn from and with the intention of throwing it away.
- The Spiral Model supported the teams in distributing the effort more evenly over phases. More specifically, it supported them in starting more actively by spending available time efficiently already from the start of the project.
- It was difficult to use the Spiral Model as an explicit means to manage time resources during the project.
- Risk analysis was mainly based on intuition and experience. No detailed, quantitative analysis of risks was made by any of the two teams.
- During risk management, new risks were identified and the conception of the original risks was modified and changed.
- The spiral is, in some situations, a misleading metaphor for a development process; in several situations, the teams found it necessary to initiate new spirals in parallel to deal with emerging problems and new risks.

Both teams were highly motivated and experienced in project work. Under these conditions, the Spiral Model was, in summary, evaluated as a useful framework for software project management, even if some minor questions suggesting further development of the framework were raised.

4. Theoretical reflection

Software development projects of any reasonable size should use variations or combinations of approaches based on specifications and prototypes. This advice is supported by the experiments and other sources, cf. (Boehm *et al.* 1984; Davis 1982). But why is this so? If this general advice is the solution, then what is the problem? What are the fundamental challenges and conditions for software development that make this advice valid and useful? In the following, we will introduce a theoretical framework and a fundamental principle for software development that provides us with general answers to these questions.

4.1. A simple model

The UCLA experiment was designed to learn about the relative strengths and weaknesses of different approaches to software development. The inquiry was based on a distinction between specifying and prototyping as two broad categories of approaches, cf. (Boehm *et al.* 1984). This distinction suggests a simple relation between the situational characteristics and the recommended approach of a software project.

In some situations, software development is based on a more or less formal *specification* of requirements. This specification is then transformed into the final system through a number of phases. In each phase, a new specification or description is developed through transformation of the description produced in the previous phase. The use of specifications is closely related to an analytical mode of operation. Systems developers are advised to take advantage of abstraction to reduce complexity. This approach has some basic limitations. It relies primarily on available information, it assumes that the available information is reliable, it implies serious simplification, and it restricts the ways in which organizational actors can communicate and learn about the future system. There have been attempts to modify the use of specifications in a more experimental manner. However, the basic problems still remain.

In other situations, software development is based on design, implementation, and evaluation of *prototypes* modeling part of the total system. The design of the system is then developed based on more or less realistic use of the prototypes. This prototyping approach to software development is closely related to an experimental mode of operation. Systems developers are advised to take advan-

tage of the possibility of learning through experiments. The prototyping approach is a constructive response to some of the problems and weaknesses of the specifying approach. But with the emphasis on prototypes, learning, and involvement of users other types of problems arise.

Considering specifying *versus* prototyping and only including this one distinction leaves us with a simple and one-dimensional understanding of approaches to software development. This framework is too simple.

4.2. Reframing the issue

The practical and basic question we are concerned with is: "Which approach to software development should we use in which situation?" In the following, we propose a more elaborate, but still quite simple framework for understanding and explaining the relation between approaches and situations in software development, cf. Mathiassen *et al.* (1990, 1992). The basic concepts of this framework are illustrated in figure 4.

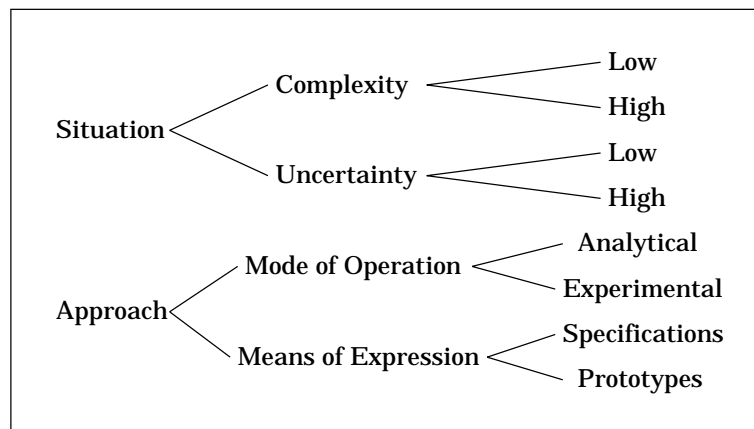


Figure 4. Situational characteristics and basic approaches to software development.

Software development situations are, on the one hand, described in terms of the degree of complexity and the degree of uncertainty that the systems developers are facing. As noted above, the degree of complexity represents the amount of relevant information that is

available in a given situation as a basis for making design decisions. In contrast, the degree of uncertainty represents the availability and reliability of information that could be relevant for the same purpose.

Approaches to software development are, on the other hand, characterized in terms of their basic mode of operation and means of expression. The mode of operation defines how systems developers are advised to process information in order to make design decisions. The mode of operation may, in one extreme, be analytical and, in the other, experimental. When systems developers operate in an analytical mode they simplify the available information through abstraction. In contrast, when operating in an experimental mode, they learn from experiences thereby generating new information.

In addition, each approach is characterized by the means of expression that are used for describing and documenting design proposals and decisions. Specifications can be used as means of expression to abstractly describe the properties and behavior of a system. As opposed to this, different models such as prototypes and mock-ups can be used as means of expression to illustrate the concrete behavior of a system.

In the simpler framework based on the distinction between specifying and prototyping we tend to take for granted that an analytical mode of operation and the use of specifications go hand in hand just like experimentation and prototypes, cf. (Mathiassen *et al.* 1990, 1992). We also tend to relate the two basic approaches to each their situational characteristic, i.e., specifying is considered effective when facing complexity, and prototyping when facing uncertainty. A fundamental premise for such a theory is that complexity and uncertainty are independent characteristics of a design situation. It is, however, difficult to find evidence supporting this viewpoint. On the contrary, behaving in an analytical way we have to rely on an imaginary simplified world, thereby introducing new sources of uncertainty as to what extent this view is in accordance with the complex real world. Correspondingly, behaving in an experimental way we produce information as we go along, thereby introducing new sources of complexity.

In our more elaborate framework, illustrated in figure 4, the assumption is that complexity and uncertainty are intrinsically related. As a consequence, there is no simple way of relating means of expression to modes of operation. When we consider complexity and

uncertainty as closely related, we cannot hope to reduce one of these without affecting the other. This is expressed in the following basic principle of software development, cf. (Mathiassen & Stage 1990, 1992):

The Principle of Limited Reduction: Relying on analytical behavior to reduce complexity introduces new sources of uncertainty requiring experimental countermeasures. Correspondingly, relying on experimental behavior to reduce uncertainty introduces new sources of complexity requiring analytical countermeasures.

The Principle of Limited Reduction describes the relationship between a situation and the mode of operation applied. It does not take into account the different means of expression. Instead, it is suggested that application of any of the two basic means of expression require a certain mixture of an analytical and an experimental mode of operation. Plans are examples of analytical countermeasures to an experimental approach based on prototypes. Likewise, quality assurance activities such as walkthroughs and reviews are examples of experimental countermeasures, performed intellectually rather than practically, to an analytical approach. They are designed to compensate for the sources of uncertainty, introduced through abstraction and specification, by exploring issues like: Is the proposed design a useful and sound basis for implementation and maintenance?

Another example illustrating the above principle is provided by the radical view of software design recommended by Parnas *et al.* (1986): They argue that the analytical mode of operation should be considered only as an ideal for software development. In practice, descriptions and specifications have to be developed, reviewed, extended, and modified in an experimental mode.

4.3. Qualitative interpretation of the experiments

In both the UCLA and the AU experiments, the task of the software teams was to develop an interactive computer system supporting the COCOMO software development cost estimation model. Even though the COCOMO model contains many parameters (16) and complicated procedures, the problem domain is quite structured and the complexity of the task is at most moderate. In designing the key data structures and algorithms, the project teams got substantial support from the COCOMO model. The main challenge is the design

of the user interface. In addition to requirements related to functionality and robustness, the system had to be easy to learn and easy to use.

Considering the task of the projects, we conclude that the complexity was moderate while the uncertainty was somewhat higher. In the specific situations of the involved projects, other characteristics were important as well. But this general analysis suggests that an effective approach should include experiments with prototypes as a key element. A traditional approach based on specifications is not suited to the challenges at hand. This provides one interesting explanation of the observed differences between the performance of the specifying approach as opposed the performance of the prototyping and mixed approaches, cf. section 3.3.

More generally, the experiments suggest that it is worthwhile to pursue the idea of combining different means of expression in the same development effort, and that the Spiral Model is a useful framework for combining specifications with prototypes, even if it requires high management competence:

- A key result of comparing the two experiments is that the mixed approaches seem to combine the strengths of the two specialized approaches. On each of the properties considered, the mixed approaches led to products of a quality that was at least comparable to the products of the two specialized approaches.
- A key result of the AU experiment is that the Spiral Model provides a useful framework for combining specifying and prototyping approaches in software development. The Spiral Model is, however, not a simple procedure to be followed. It is rather a general framework for understanding and managing software projects and it is quite open to individual interpretations.

The experiments have, in this way, illustrated the practical advice implied by the Principle of Limited Reduction: Analytical and experimental modes of operation should not be understood and used independently of each other. Effective software design requires a systematic effort combining analytical and experimental modes of operation.

Acknowledgment

This research has been partially sponsored by the Danish Natural Science Research Council, Program No. 11-8394. We wish to thank Troels Larsen, Sanne Liebmann, Casper Millum, Helge Solberg, and Frank Tolstrup for their effort and cooperation during the experiment. In writing this article, we have received valuable comments and suggestions from Barry Boehm, Kaj Grønbaek, Karlheinz Kautz, Andreas Munk-Madsen, Peter Axel Nielsen, Carsten Sørensen, Ivan Aaen, and the three anonymous reviewers.

References

- Andersen, N. E., F. Kensing, J. Lundin, L. Mathiassen, A. Munk-Madsen, M. Rasbech & P. Sørgaard (1990): *Professional Systems Development. Experience, Ideas, and Action*. Englewood Cliffs, New Jersey: Prentice-Hall.
- Boehm, B. W. (1981): *Software Engineering Economics*. Englewood Cliffs, New Jersey: Prentice-Hall.
- Boehm, B. W. (1988): A Spiral Model of Software Development and Enhancement. *Computer*, May.
- Boehm, B. W., T. E. Gray & T. Seewaldt (1984): Prototyping versus Specifying: A Multiproject Experiment. *IEEE Transactions on Software Engineering*, Vol. 10, No. 3 (290–303).
- Budde, R., K. Kautz, K. Kuhlenkamp & H. Züllighoven (1992): *Prototyping—An Approach to Evolutionary Systems Development*. Berlin: Springer-Verlag.
- Davis, G. B. (1982): Strategies for Information Requirement Determination. *IBM Systems Journal*, Vol. 21, No. 1 (4–30).
- DeMarco, T. (1979): *Structured Analysis and System Specification*. Englewood Cliffs, New Jersey: Yourdon Inc. & Prentice-Hall.
- Dijkstra, E. (1972): Notes on Structured Programming. (1–82) in: *Structured Programming*. London: Academic Press.
- Fairley, R. (1985): *Software Engineering Concepts*. McGraw-Hill.
- Floyd, C. (1984): A Systematic Look at Prototyping. (1–17) in R. Budde *et al.* (Eds.): *Approaches to Prototyping*. Berlin: Springer-Verlag.
- Floyd, C. (1987): Outline of a Paradigm Change in Software Engineering. (191–210) in G. Bjerknes *et al.* (Eds.): *Computers and Democracy*. Avebury: Aldershot.
- Gomaa, H. & D. B. H. Scott (1981): Prototyping as a Tool in the Specification of User Requirements. (333–342) in: *Proceedings of the 5th IEEE International Conference on Software Engineering*.

MIXED APPROACHES

- Gould, J. D. & C. Lewis (1985): Designing for Usability: Key Principles and What Designers Think. *Communications of the ACM*, Vol. 28, No. 3 (300–311).
- Jepsen, L. O., L. Mathiassen & P. A. Nielsen (1989): Back to Thinking Mode—Diaries as a Medium for Effective Management of Information Systems Development. *Behavior and Information Technology*, Vol. 8.
- Langefors, B. (1966): *Theoretical Analysis of Information Systems*. Lund: Studentlitteratur.
- Larsen, T., S. Liebmann, C. Millum, H. Solberg & F. Tolstrup (1990): *The Spiral Model Used in Practical Systems development*. Ms.S. thesis, Institute for Electronic Systems, Aalborg University. (In Danish)
- Mathiassen, L. & J. Stage (1990): Complexity and Uncertainty in Software Design. (482–489) in: *Proceedings of the IEEE International Conference on Computer Systems and Software Engineering*. Washington DC: IEEE Computer Society Press.
- Mathiassen, L. & J. Stage (1992): The Principle of Limited Reduction in Software Design. *Information, Technology & People*, Vol. 6, Nos. 2–3 (171–185).
- Parnas, D. L. & P. C. Clements (1986): A Rational Design Process: How and Why to Fake it. *IEEE Transactions on Software Engineering*, Vol. 12, No. 2 (251–257).
- Pressman, R. S. (1987): *Software Engineering. A Practitioner's Approach*. Maidenhead, Berkshire: McGraw-Hill, second edition.
- Stage, J. (1989): *Between Tradition and Transcendence. Analysis and Design in Systems development*. Ph.D. thesis. Institute for Electronic Systems, Aalborg University. (In Danish)
- Wirth, N. (1973): *Systematic Programming. An Introduction*. Englewood Cliffs, New Jersey: Prentice-Hall.
- Wulf, W. (1977): Languages and Structured Programs. In R. T. Yeh (Ed.): *Current Trends in Programming Methodology*. New Jersey: Prentice-Hall.
- Yourdon, E. (1982): *Managing the System Life Cycle*. New York: Yourdon Inc.