Problems with software production

ESSENCE AND ACCIDENTS — BROOKS' POINT OF VIEW

The case of hardware

- Recall the definition of the performance-price factor on transparency 24:

  performance-price factor $\stackrel{def}{=}$

  time to perform 10^6 additions $\times$ cost of CPU and main memory

- Over the past 30 years hardware has become cheaper and faster — the performance-price factor has decreased by an order of magnitude six times!

- Will this trend continue?

- No, due to constraints of physical nature no more than 2 such order of magnitude decreases are anymore possible.

- Why?
  - The *speed of light* sets an upper limit to the propagation speed of electro-magnetic waves. Consequently, the only way to move forward is miniaturization of chips
  - An electron can travel along a path as narrow as 3 atoms but not less narrow than that, because it then strays to an adjacent path.
  - Putting paths too closely together leads to interference between paths.

- Consequently *there are intrinsic laws of nature preventing computers from becoming arbitrarily fast or arbitrarily small.*

CRITIQUE: The conclusion seems to be correct. However, possibly altogether different principles of building computers may lead to progress on this issue. For nature developed its own super-computer: *the*

human brain, and isn't that by far more efficient than present day hardware in size and speed?

PROBLEMS WITH SOFTWARE PRODUCTION: BROOKS' POINT OF VIEW

- Software is essentially conceptual, i.e. an artifact, and therefore non-physical (although it is of course always stored on a medium). So, superficially, any increase in productivity and quality seems to be possible.

- Brooks, in his 1986 paper *"No silver bullet"*, claims such increases are only possibly to a very limited extent.

  The "silver bullet" in the title of Brooks' article refers to the recommended way of slaying werewolves, otherwise perfectly normal human beings who suddenly turn into wolves. Brooks' line of inquiry is to determine whether a similar silver bullet can be used to solve the problems of software. After all, software usually appears to be innocent and straightforward, but like a werewolf, software can be transformed into

something horrifying, in the shape of late deadlines, exceeded budgets, and residual specification and design faults not detected during testing.

- *Brooks argues that, analogous to the limits to hardware speed and size, there are inherent problems with current techniques of software production that can never be solved.*

- He distinguishes between 4 essential problems with such techniques, called *essence — the difficulties inherent in the intrinsic nature of software —*, and *accidents*, the difficulties *that can be overcome today* which are not inherent in software production.

- These 4 essential problems are discussed below. They are:
  - the **complexity** of software
  - its **conformity**
  - its **changeability**
  - its **invisibility** (or rather **invisualizability**)

Complexity of software

The problem:

- Take an automaton with 10 states and consider the parallel composition of 30 of such automata. This results in a product automaton of $10 \uparrow 30$ states — that is, more states than the number of electrons within our universe!

- Consider now the INTERNET. Certainly only 30 computers operating in parallel is chicken feed — many many more are operating in parallel! As we saw above the complexity of the product automaton is **exponential** in the number of constituting automata. Consequently, a description of the INTERNET using an enumeration of in principle possible states is COMPLETELY IRREALISTIC.

What can be done about this?

- An analysis using product automata from automata theory may NOT be the appropriate one to analyze the parallel composition of programs. After all, human beings operate in parallel, so Darwin's law "survival of the fittest" dictates they do so efficiently — but NOT by enumerating (invisible) state changes inside human beings and then constructing the product space!

- As possible solution so-called **compositional specification methods** have been suggested. These yield (formal) specifications of processes **regardless** *of the environment these processes operate in*, or, under certain minimal assumptions about their environment.

- Using compositional specification methods, the parallel composition of processes can be specified using the *conjunction* of the specification of the separate processes. Hence the complexity of this description is **linear**, rather than exponential, in the number of processes.

- However, using de Morgan's law about the

distribution of disjunctions over conjunctions such a linear description may still, in the extreme, lead to an exponential number of separate cases to be considered. So what did we gain?

- Return to the analogy with mankind, above. What other (wo)men observe are not the state changes inside their fellow human beings but rather changes in their **(inter)faces** or attitudes from which a lot of local internal changes have been eliminated.

- Similarly, compositional specifications record changes in interfaces between programs (or processes), i.e. the change of externally (i.e. to other programs) visible quantities, and NOT the change in internal quantities s.a. local variables, local channels and the like. Also compositional specifications express the **condition** under which such external changes occur.

- The latter is extremely important. For de Morgan's law still applies w.r.t. distributing disjunctions inside (compositional) specifications over conjunctions

between specifications, still resulting in a, hopefully considerably diminished, but still exponential number of cases to be considered. However, due to the fact that compositional specifications also express the conditions under which externally visible changes occur, it becomes usually immediately obvious that only *a few of such combinations of externally visible changes* (inside separate processes) *actually occur*, i.e. are *consistent* — one hopes at least so few that the number of consistent combinations has linear complexity.

- The assumption behind this phenomenon is that communicating computers, operating in parallel, are invented by man, and, since mankind cannot cope with exponential complexity, anything man-made $which\ works$ and consists of parallel components must somehow be possible to characterize using a very much lower than exponential complexity in the number of processes.

- The compositionality paradigm above has been mentioned in order to explain that it may be possible

(in the further future) to characterize the complexity of man-made networks of processes by a very much lower than exponential complexity (in the number of processes), since these networks are man-made artifacts.

Conformity of Software

- **Example:** A manually controlled gold refinery is to be computerized. A computer will send the necessary control signals to the components of the existing plant, instead of using humanly operated levers and buttons.
  - The task of the software development team is to construct a product interfacing with the existing plant.
  - I.e., THE SOFTWARE MUST CONFORM TO THE PLANT, NOT THE PLANT TO THE SOFTWARE.
    This is an example of Brooks' style conformity.

- What if a brand new computerized gold refinery were to be constructed by a team in which mechanical, metallurgical and software engineers cooperate?

  In practice, it is considered easier to make the software interface conform to the other components,

rather than **change** the way these have been configured in the past.

- The problems caused by this forced conformity is not due to the structure of the software itself but by the externally imposed structure on, and by, their interfaces. Hence its complexity cannot be removed by improving the software process.

CRITIQUE:

- It is said that software systems are the most complex constructions ever built by man. This is not said of the control of plants. Hence their control cannot be so terribly complicated.

- Furthermore, the fallacy underlying the gold refinery example is a common one. This example extends to control of arbitrary plants. Now it is simply NOT TRUE that the structure of such plants is calculated by control engineers. There is no such thing!

- Their structure is mainly based upon empirical

knowledge, and *only very partially on mathematics at all!*

In reality, the design of an existing successful plant is taken and then extrapolated by small steps to find out which extensions can take place, leading to a balanced design which functions indeed.

- THERE SIMPLY DOESN'T EXIST ANY KNOWN BODY OF MATHEMATICAL KNOWLEDGE in which constructions of arbitrary complexity can be calculated, or, using which one can synthesize implementations from specifications of arbitrary complexity!

- That such could be done is a typical fallacy assumed by post-modern clients (politicians, planners) in the last quarter of this century, as testified by:
  - the DISASTER of constructing the "Klappbrücke" over the "Kieler Förde"
  - and the many other software DISASTERS, e.g., concerning the construction of the Denver airport automatized luggage control system, or that

concerning the USA civil airplane surveillance and routing system, mentioned above (a journal exists which is devoted to listing these software disasters), and so on.

Changeability of Software

- It is unreasonable to ask a civil engineer to move a bridge over 100 miles, or to rotate it through 90^o, but it is perfectly acceptable to tell a software engineer to rewrite half an operating system over a 5-year period.

- Now, software engineers are well aware that, in the long run, extensive maintenance is unwise and that rewriting the product from scratch will sometimes prove to be less expensive. Nevertheless, clients frequently demand *major changes* to software.

- As Brooks points out, there are always pressures to change (and develop) existing software. *This is an essential, not an accidental property of the software development process*; after all, it *is* easier to change software than, say, the hardware on which it runs; that is the reason behind the terms SOFTware and HARDware.

- There are 4 reasons why useful software will always undergo change:

  1. Software is a model of reality, and reality changes.
  2. If software is useful, there are pressures (chiefly from satisfied users) to change the functionality of the product BEYOND WHICH IS FEASIBLE IN TERMS OF THE ORIGINAL DESIGN.
  3. Software is so much easier to change than hardware.
  4. Successful software lives far beyond the hardware for which it is written — it has to be modified to some extent to run on the new hardware.

Hence, it is part of the ESSENCE of software that it has to be changed, and this inexorable and continual change has a deleterious effect on its quality.

CRITIQUE: NONE. **Changeability of software is inescapable**, one of its most essential properties.

Invisibility of Software

- Brooks' fourth major problem with software is that it is

  "invisible and unvisualizable"

  Example: Obtain a 150-page listing of object code and try to modify aspects of it.

- In contrast, architects build scale models or make blueprints, chemists build models of molecules, plastic surgeons use the computer to show their clients how their faces will look like after surgery (and recovery). VLSI chips are represented by various kinds of schemata, at different levels of abstraction.

- Now visualizations can be made of **aspects** of software, i.e., *control flow, data flow*, illustrating *dependencies between modules*, or using *timing diagrams* to depict causality over time in a product.

However, as the critique goes, the resulting graphs *are not planar*, let alone, **hierarchical**. They can only depict a **subject** of the product in visually acceptable fashion.

- The main critique here on such visual representations is that *diagrams cannot embody* EVERY aspect of the product — its object code must always be consulted to analyze its efficiency, e.g. — nor is there a *generally accepted way to determine what's missing from such diagrams*.

CRITIQUE:

- Although the above is true, it's also the most dated one concerning essential problems with improving quality and efficiency of development of software. E.g., David Harel's STATEMATE constitutes a major breakthrough in constructing a *visually attractive executable specification of software* in the area of *real-time embedded systems* (this system will be subject of the "Fortgeschrittenen-Praktika" of my chair, coming semesters), which is accepted by

the industries for which it is built (mainly aerospace industry).

Conclusion of Brooks' "No silver bullet"

- Brooks makes a *distinction* between the *essential problems* in software production — called their *essence* — and *accidental* problems which can be overcome using some new software tool.

- Consequently, e.g. *timesharing, the UNIX Programmer's Workbench, ADA, software development environments, proofs of correctness, object-oriented design, expert systems, new software engineering techniques*, are all classified as solving only accidental problems.

- This is so because they do not lead to an order of magnitude (10 x) improvement in quality and productivity within $\pm$ 5 years.

- Instead, the past 20 years have shown a steady productivity increase of 6 % per year in software production, that is, a doubling of productivity in 12

years (but not of its quality - see e.g. WINDOWS 95).

- Brooks concludes that an order of magnitude increase, i.e. a "silver bullet" is unlikely.

CRITIQUE:

- Brooks' conclusion seems to be still right; comparatively speaking, building a 2-litre-fuel-consumption-per-100-kilometres car seems a very unlikely possibility. But that's also its main critique! Is there really a difference between *essential and accidental problems* in software production? Isn't it rather the case that setting irrealistic targets is a much more wide-spread problem (due to irrealistic practices of politicians, managers and so on), see, for instance the long lists of the resulting disasters mentioned previously? And that, especially in case of software production, targets tend to become irrealistic because the planners involved have simply no notion of the structure and COMPLEXITY of the required product.

Life-Cycle Models

Objectives

In this Chapter you will:

- Learn about four different life-cycle models, namely the build-and-fix model, the waterfall model, the rapid prototyping model, and the spiral model.
- Realize that the build-and-fix model should be avoided because it costs too much.
- Appreciate that the waterfall model has many successes, but that the product delivered to the client may not be what the client really needs.
- Understand that the rapid prototyping model ensures that the client's real needs are met.
- Learn when to use the risk-driven spiral model.

- The life-cycle model is the series of steps through which the product progresses.

- Without a preplanned life-cycle model, the members of the development team will work aimlessly.
  The life-cycle model is like a game-plan on which all members of a team can agree before development commences.

The Build-and-Fix Model

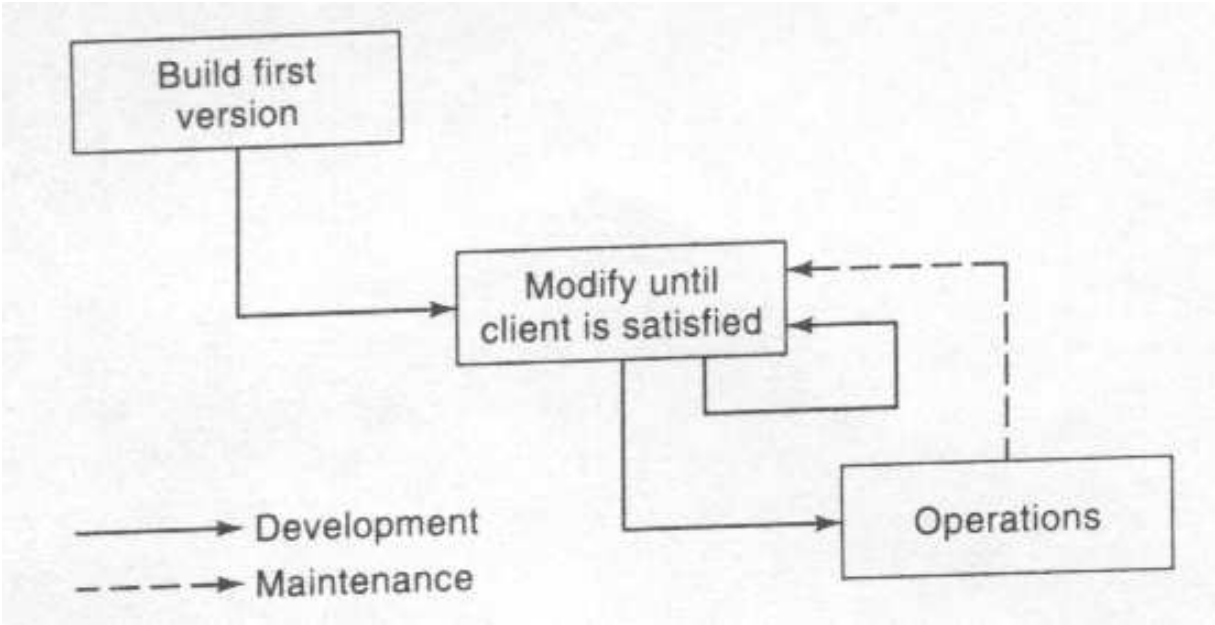

Figure 1: Life-cycle phases of build-and-fix model

- The build-and-fix model is depicted in fig. 1, and consists of the following:
  - The entire product is built and delivered to the client.
  - The client points out what has to be changed, and changes are made until the client is satisfied.

  - Then the product is used productively by the client.

- *The build-and-fix model should be avoided at all costs.* This has the following reasons:
  - The *high cost* of *making the changes* which the user requires *after the complete software product has been built.*
  - For a specification fault is 50 times cheaper to fix while the specification document is being drawn up than when it is corrected during maintenance phase, running on the client's computer.
  - The build-and-fix model does not provide any *coherent and cohesive overall* structure. Consequently maintenance becomes a nightmare.

- Consequently, the build-and-fix model is NOT COST-EFFECTIVE over the whole lifetime of the product.

- There is only one instance when to use the the build-and-fix model: if the product is small and does

not require maintenance, i.e., for a small student assignment (NOT FOR THIS PRACTICUM!!).

The Waterfall Model

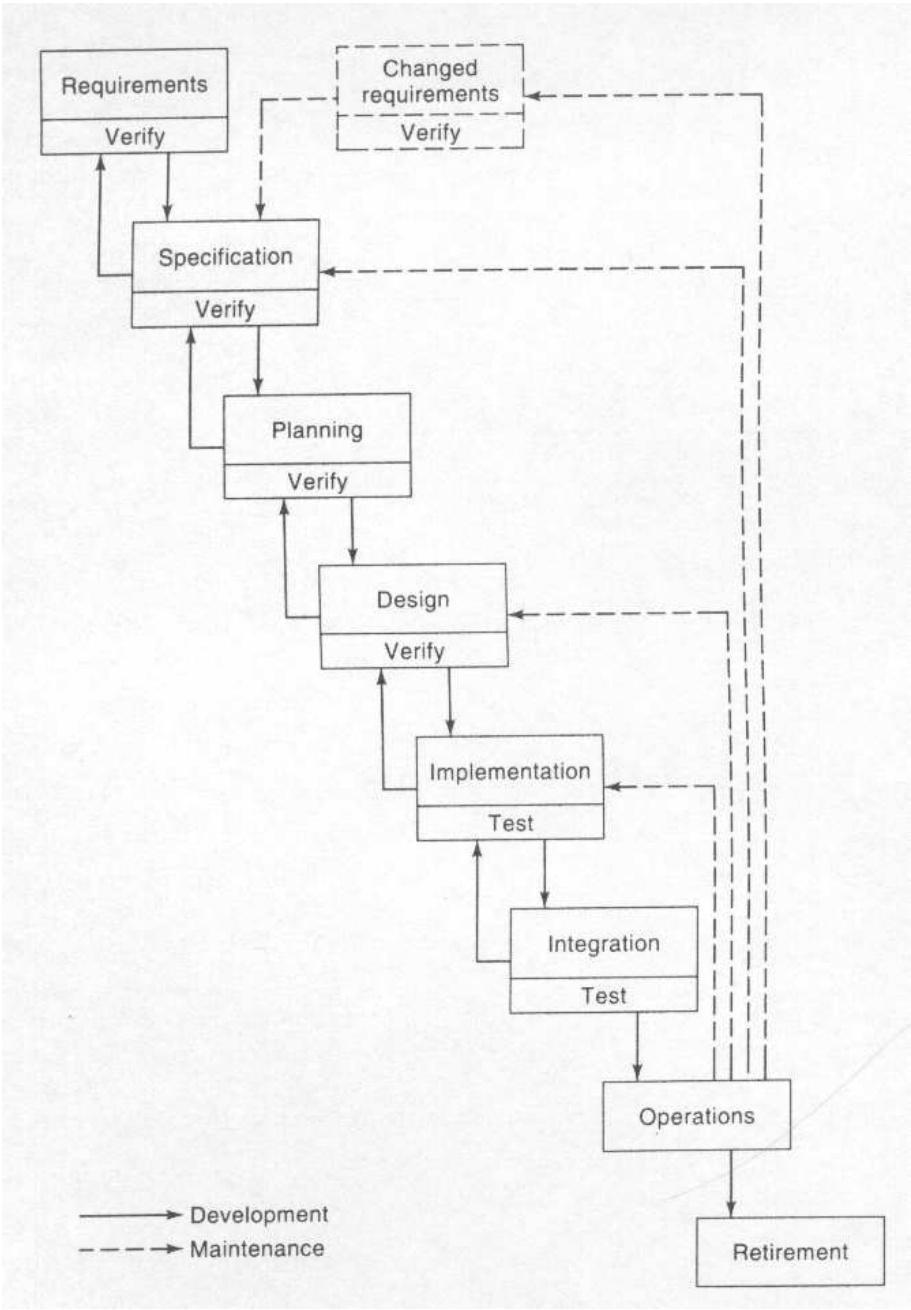

Figure 2: Life-cycle phases of waterfall model

- The waterfall model is depicted in figure 2 above.

- The waterfall model has been in use for more than 20 years. It defined for the first time a proper full life-cycle of a software product. It has emerged the hard way in the 70's after many extremely costly mistakes made by big software companies, notably IBM, in the 60's.

- Each of the requirements, specification, planning and design phases has been described before (see transparencies 40–60) and is followed by a $Verification\ Phase$ to be carried out by the Software Quality Assurance team.
  The role of testing and that of the SQA team have been explained on transparencies 17–20.

- Similarly, the implementation, integration and maintenance phases have been described on transparencies 60–68 as is the rôle of the testing phase after each of these phases.

- THE BASIC PRINCIPLE UNDERLYING THE WATERFALL MODEL IS THAT, EVEN IF

WE DO OUR BEST AT EVERY PHASE, THERE WILL ALWAYS EVENTUALLY OCCUR A NEED TO BACKTRACK AND CORRECT THE WORK OF A PREVIOUS PHASE.

Strengths and Weaknesses of the Waterfall Model

- The major reason for the success of the waterfall model is that it is $DOCUMENT\text{-}driven$; no phase is deemed complete until the documentation for that phase has been approved.

- For the primary cause of failure of many software projects is inadequate documentation, or worse:
  no documentation, or still worse:
  incorrect documentation!

- Another strenght of it is that one should try to get things right the first time.
  However, if one fails – and such a moment always eventually occurs – the feedback loops *provide the mechanism for correcting the situation.*

- However, its main strenght is also its main weakness in other respects: the client is provided with a

written specification document. If it is for software professionals difficult to visualize the functioning of a software product, *how much more difficult must this be for a client, often a computer illiterate?*

- In other professions this is different: architects provide scale models before starting to build, as do engineers before manufacturing.

- Consequently, the waterfall model is *less suited* to find out *the real need of a client*, as opposed to the needs he/she says to have.

- The solution to this problem is the: *Rapid Prototyping Model.*

The Rapid Prototype Model

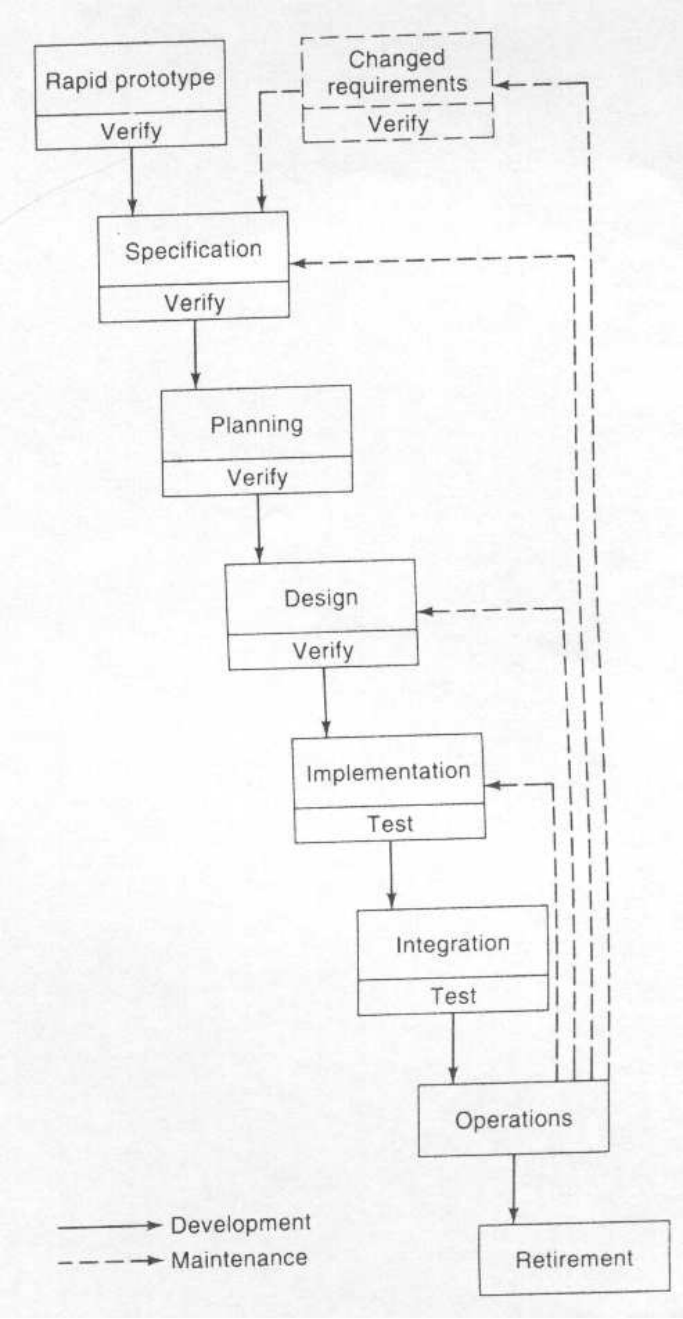

Figure 3: Life-cycle phases of rapid prototyping model

- The rapid prototyping model will be illustrated extensively in the case study discussed a few pages from now. Its main change with the waterfall model is that there is much less emphasis on feedback loops. This is o.a. due to the fact that due to using a rapid prototype the requirements phase is much better and satisfactorily executed; consequently, *the rapid prototype indeed reflects precisely what is needed.*

- A rapid prototype is a program put together quickly, for displaying the desired (I/O) functionality to the client so he/she can play with it until the desired functionality has emerged, and not the functionality which he/she stated in the beginning (during concept exploration). Consequently, it must be easy to modify this prototype.

- A specific verification phase follows the Rapid Prototype phase in which the SQA team checks that (representatives of) the prospective users have approved the functionality of the prototype.

- Since the rapid prototype reflects precisely what the client and user need, there is no need for a feedback loop from the specificaion phase. Similarly, since the development team has already gained insight in how to design the final product from the experience of using the prototype, a feedback loop leading from the design phase is *less likely* to be needed.

- The problem with feedback loops is that they consume valuable time, and the later in the life-cycle a loop occurs, the more time it consumes. *By largely eliminating the need for feedback, the rapid prototype model speeds up the software development life-cycle.*

The Spiral Model

- The spiral model can be used only if two criteria are met:
  - *The software to be developed should be large.*
  - *The product must be developed internally, e.g., by IBM for IBM* (IBM introduced it for the first time).
- The spiral model is depicted in fig. 4 below.

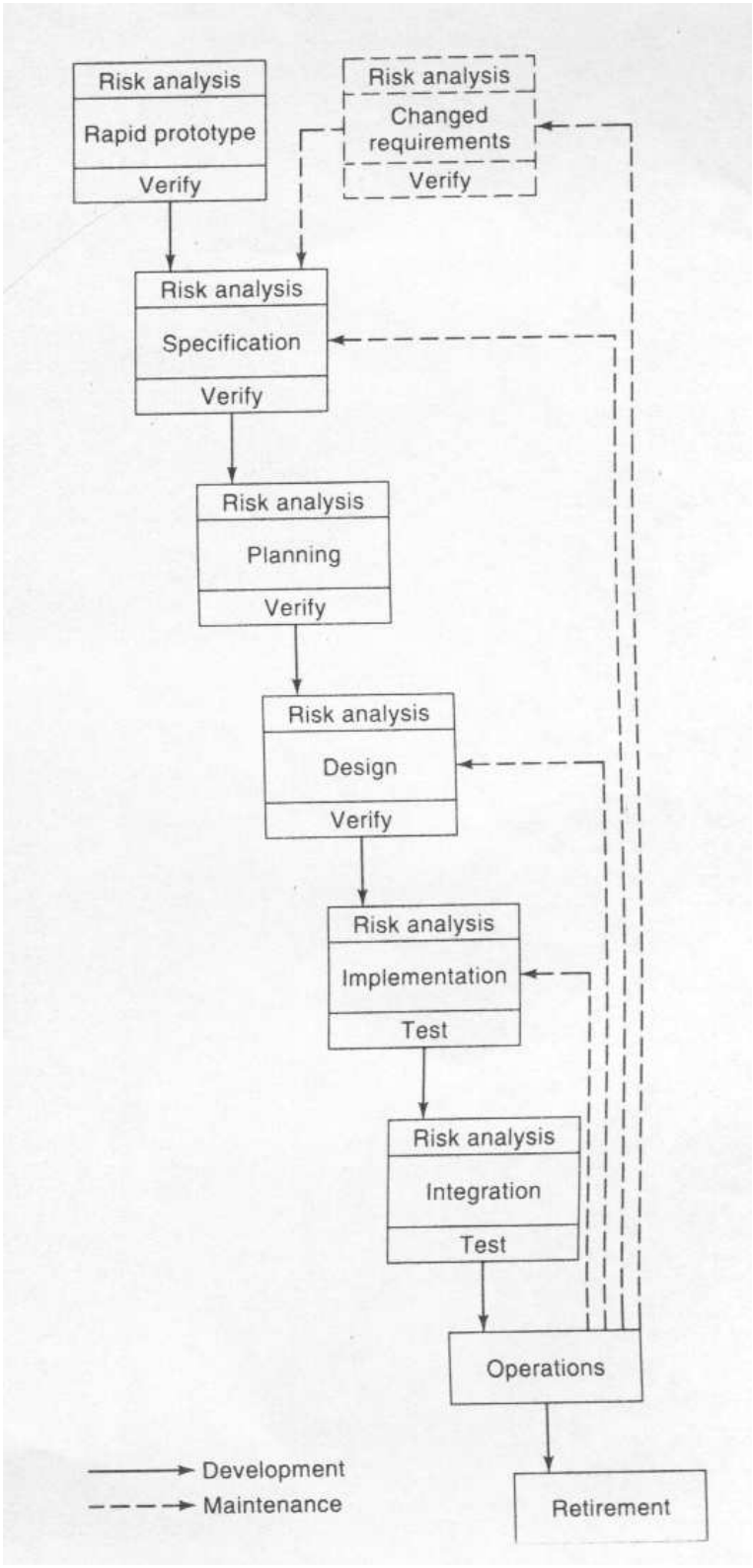

Figure 4: Life-cycle phases of spiral model

- The main difference between the rapid prototyping model and the spiral model is that RISK ANALYSIS takes place at every phase of it. Consequently, it is possible to cancel the project at every phase.

- The possibility of cancellation after every phase makes this model unworkable for software projects contracted with an external client. For, after the specification and planning pahses, after client and software development team have agreed and the contract signed, any breach out of that contract tends to lead to a lawsuit.

- In general the principle of risk analysis is an important one. For, whether the contract is internal or external, it is always necessary that *the client performs a detailed risk analysis before signing a contract.*

Case Study: Risk Analysis

- Recall our case study (on transparencies 26–27). It is outlined on transparency 110.

- Before engaging the SRS Software Engineering company, Mr. Chesterton consults a management consultant Arabella Swinson regarding the risks involved in building the management-by-objectives product.

- *"How can our risks be minimized?"*, he asks

- *"If sales continue at the present level your company is bankrupt within 2 years"*, is the reply, *"So it is essential that the product be developed.* Moreover its costs can be easily borne by you, when reflecting the expected increased turn-over. After all, nobody wants to loose his/her job."

- "So you should immediately implement the management-by-objectives schemes manually, to fall

back upon if the product is late or doesn't function, and both for the region managers to get experience in estimating sales targets for each month and for the shop managers to learn to motivate personnel to meet those targets".

- "You should try to negotiate penalty clauses in case the product isn't delivered correctly on time. If SRS doesn't agree to this, you should insist on including a clause that no money is paid until the product has been delivered and found to perform according to the specification document. This will speed SRS's work up, especially in case you promise to pay a bonus in case of early delivery."

Homework Assignment

- Select a life-cycle model for developing the ChocAn product. Give careful reasons for your choice.

CASE STUDY: Requirements Phase

- In this case study you will:
  - Discover that rapid prototyping is an excellent requirements technique.
  - Learn how rapid prototyping is used in practice.
  - Be warned about the potential pitfalls of rapid prototyping.

- Recall that during the requirements phase, the real needs of the client are being determined (using system analysis techniques), rather than just adopting bluntly what the client says she/he thinks she/he needs.

- In the case study used, rapid prototyping is used to determine the clients' requirements.

- Recall transparencies 26, 27 for our case study; its main characteristics are:

- OUTLINE:
  - A chocolate-coated-chocolate tycoon called Chesterton — head of the C-CC corporation — has 27 shops for selling his product in and around New York.
  - Productivity has decreased and a management consultant has been called in, Arabella Swinson.
  - She suggests to implement *management-by-objectives*.
  - I.e. every shop is assigned a **sales target** each month; the manager then encourages every employee to reach that target.
  - In order to keep track of how well each shop is performing relative to its sales target, Mr. Chesterton engages ESSARES Software Engineering to computerize the management-by-objectives scheme.

The Rapid Prototype Case Study

Mr. Chesterton explains to Chip Brokenshaw, a prototyper at Essares (SRS), how *management-by-objectives* works:

- Each of his **27 shops** is assigned a sales target for each month.
- Effectivity in reaching this target is measured, according to Chesterton, using the following concept:

$$\text{Effectiveness} \stackrel{def}{=} \frac{\text{actual sales}}{\text{target sales}} * 100\%$$

- In case "Effectiveness" is less than 100 % over a month, the computer should send a report to the **shop manager**, stating that *the objective for that month has not been met*.

Otherwise, if "Effectiveness" is ≥ 100 %, the report states that the objective for that month *has been achieved*.

- All reports must be sent to the relevant **region manager** (Chesterton has shops in New York -NY-, Delaware -DE-, Pennsylvania -PA-) with copies to the **vice-president for sales** of the C-CC corporation.

- The C-CC corporation has 27 shops in 3 states: 15 in NY, 8 in PA, 4 in DE. Each state constitutes a **sales region** with a region manager; the 3 region managers are subordinate to the vice-president for sales.

- These 4 people should, apart from all the shop managers, receive monthly progress reports stating **the target data for a given shop for the whole year**, and **the actual sales from January up to the current month**.

- Then Chip B. announces he'll construct a rapid prototype, i.e., he explains that he'll write a program

that won't do everything the complete product does, but that will do the key things (*here B. makes an error*).

- Subsequently Chesterton asks: *"Which programming language are you going to use for the prototype?"*

- Brokenshaw answers: "Smalltalk, Prolog, Hypertext or another 4th generation language."

- Chesterton answers: "My son-in-law learned C at college. C is the language of the future. So C should be used throughout. "

- *(Brokenshaw's 2nd error)*: B. gives in, because he needs the project.

- Then B. leaves stating he'll return next day with a prototype written in C (it is shown in appendix B of Schach's *Practical Software Engineering, IRWIN publ. comp.*) (here he could have corrected his 1st error.)

- B. returns next day and shows a sample report:

C-CC Shop Report

Shop No.: 132

Address: 47 Cloudland
Wissimack, DE 19989-0089

Manager: Jaime Morris

Telephone: (302)543-6970

Region: DE

Sales through March 1992

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
Target Sales ($K)	40	50	50	60	60	65	70	75	80	80	85	90
Actual Sales ($K)	20	40	75									
Effectiveness	50%	80%	150%									
Objectve Achieved	No	No	Yes									

Sales objective for March has been achieved

Figure 5: Output from first version of rapid prototype

- Chesterton answers critically:

    1. All headings must be in uppercase letters.

2. The first month the target was 40 K$, of which 20 K$ actually earned, the 2nd month sales doubled to 40 K$, but since the target was 50 K$, taken literally the objective has not been achieved. However, the double sales should be rewarded, and not punished, by stating *"Objective Received: No"* in February.

- Brokenshaw finds a solution *(and makes the first correction upon the SRS C-CC example specification of his client)*, by introducing the concept *shortfall*:

$$\text{Shortfall} \stackrel{def}{=} 100\% - \text{Effectiveness}$$

as a separate category for evaluation, and proposes that a shop fails its objective only if *the shortfall is more than half of that of the previous month*.

- Chesterton agrees and Brokenshaw states he'll fix the prototype and return in half an hour ($here\ B.\ makes\ another\ mistake,\ as\ we'll\ see$).

- Brokenshaw returns and displays:

C-CC SHOP REPORT

SHOP NO.: 132

ADDRESS: 47 Cloudland
Wissimack, DE 19989-0089

MANAGER: Jaime Morris

TELEPHONE: (302)543-6970

REGION: DE

SALES THROUGH APRIL 1992

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
Target Sales ($K)	40	50	50	60	60	65	70	75	80	80	85	90
Actual Sales ($K)	20	40	75	59								
Effectiveness	50%	80%	150%	98%								
Shortfall	50%	20%	--	2%								
Objectve Achieved	No	Yes	Yes	No								

SALES OBJECTIVE FOR APRIL HAS NOT BEEN ACHIEVED

Figure 6: Output from second version of rapid prototype

- Chesterton approves the column for February but disapproves the column for April, because if the shortfall is 5 % or less he finds the objectives are still reached *(this is the 2nd correction upon C's original specification).*

- Brokenshaw withdraws and produces figures 3 and 4 below:

C-CC REPORT TO REGION MANAGER

REGION: DE

MANAGER: Frank di Bella
17 Hampstead Avenue
Tarkington DE 18743-6189

TEL. NO.: (716)843-4702

SALES FOR APRIL 1992

Shop No.	Street Address	City	Target Sales $K	Actual Sales $K	Effectiveness	Achieve Objective
034	Heath Center, Shop 4	Patterson DE 19402	125	103	82%	No
132	47 Cloudland	Wissimack DE 19989	60	59	98%	Yes
174	38 Park Terrace	Markham DE 19783	132	139	105%	Yes
239	3020 Harrod Place	Congreave DE 19783	50	22	44%	No

4 SHOPS IN REGION
2 SHOPS ACHIEVED OBJECTIVE
2 SHOPS DID NOT ACHIEVE OBJECTIVE
0 SHOPS NO DATA AVAILABLE

Figure 7: Report produced by third version of rapid prototype

REPORT TO C-CC VICE-PRESIDENT FOR SALES

SALES FOR APRIL 1992

Region Name	Number of Shops in Region	Number of Shops Achieving Objective	Number of Shops not Achieving Objective	Number of Shops with No Data Available
DE	4	2	2	0
NY	15	11	1	3
PA	8	7	0	1

27 C-CC SHOPS
20 SHOPS ACHIEVED OBJECTIVE
3 SHOPS DID NOT ACHIEVE OBJECTIVE
4 SHOPS NO DATA AVAILABLE

Figure 8: Report produced by third version of rapid prototype

- He returns, shows them to C, who approves his rapid prototype and says: "When do I get the complete product?" *(Here B. makes another mistake.)*

Lessons From The Case Study

1. As a consequence of developing a rapid prototype, a number of *improvements on the original specification* given by Chesterton could be made, reflecting his real needs as opposedto the needs he originally stated. Consequently, the requirements phase has been successful.

2. What Brokenshaw built was a *rapid* prototype, i.e.:

   (a) it was built *quickly* so that product development time was not unnecessarily prolonged,
   (b) it was *easy to modify*; this is essential since that is the main criterion – it should be easy to modify so as to establish the clients' real needs, and
   (c) it reflected the essential aspects of the product, in this case the included I/O behavior, i.e. the functionality of the product, allowing the client to see for himself whether this was what he/she really wanted.

3. Once a rapid prototype has been constructed there is only one thing to do – THROW IT AWAY. WHY?

   (a) Any form of enhancement constitutes a change to a running product and is consequently 50–100 times more expensive than changing requirements or the specification.
   (b) A prototype is usually full of faults.
   (c) The *expensive* thing to do is to reuse a prototype; the *cheap* alternative is to throw it away and start the specification phase.
   (d) A rapid prototype does not incorporate any security or safety features, neither does it handle exceptions (describing how the product responds upon detecting erroneous data at run time).
   (e) Using a rapid prototype as final product compromises its future maintainability because of lack of documentation.

4. The best way to ensure that a rapid prototype is not anymore used after the requirements phase is writing it in another language than the one selected for implementing the final product.

Brokenshaw's Mistakes

1. C.B's firts mistake was that he let himself be intimidated by Chesterton in his choice of implementation language for the prototype. An interpreted language (where programs needn't be linked and loaded) allows a rapid prototype to be constructed faster and more flexibly.

2. Brokenshaw's second mistake was that he did not explain to Mr. Chesterton in sufficient depth about the *distinction* between a prototype and a production-quality implementation.
   Especially after seeing that the prototype has been changed in 30 minutes Chesterton won't understand why maintaining the product takes so much time. Also now, he won't understand why it will take that long to produce the endproduct.

3. Brokenshaw's third mistake (which relates to the one described above) is that he didn't explain to

Chesterton the importance of documentation. What Brokenshaw should have done is that he should have reported to his manager, *who should have experience in dealing with people s.a. Chesterton.* This manager would have explained *tactfully* to Chesterton that what he, Chesterton needed is a well-functioning and well documented product, and then would have presented the various *alternatives*, out of which Chesterton could then select the *most cost-effective one.*

4. B's fourth mistake was that he *didn't show* the prototype to any *shop manager, region manager nor to the vice-president for sales*, for these *real users* could have informed him better of their real needs.

Testing during the Rapid Prototyping Phase

- The primary purpose of a rapid prototype is to determine the clients' real needs. To this end it is extremely important to consult the *users* of the *final* product. (After all, the only contact with many clients is usually paying the bill, after finishing the specification phase.)

- Consequently that's the task of the SQA group during the prototype phase: *seeing to it that the users of the endproduct get sufficient possibilities to interact with the prototype.*

Case Tools for Rapid Prototyping

- *Interpreted languages* are best for rapid prototyping because the programs written in them needn't be linked, loaded and compiled.

- Many rapid prototyping languages have associated with case tools, e.g. the UNIX Shell programming language is supported by the UNIX programmer's Workbench.

Homework Assignment

- Construct a rapid prototype for the ChocAn product. What components of the functionality of the final product have you deliberately omitted from your prototype? What components have you implemented only partially? Which aspects have you accentuated, and why?