Chapter III.1
Program development

Part I of this book is an introduction to functional programming using Clean. In part II a number of case studies were presented. Each of these case studies treats a concrete application or how functional programming can be used is a specific branch of computer science. In part III we will give you guidelines to construct your own applications. In this chapter we discuss the ways to build your own applications. The next chapter shows you different styles to construct your functions. Chapter 3 teaches you to understand and improve the efficiency of your applications.

1.1 Software engineering

The branch of computer science that studies the methods to construct programs and to control the software production process is called software engineering, see e.g. [van Vliet 93, Sommerville 92]. The software process[1] ranches from determining the requirements for the program to introduction of the program to users and maintenance of the product. The actual writing of the program is only a small part of this entire process. Although a complete course software engineering is outside the scope of this book, we give a brief overview of the field since this helps you also to construct small and medium sized programs.

In the construction of programs there is not a single way to success, nor a way that guarantees success. However, a well designed software process reduces the risks and signals problems early. The general approach of all models is to split the software process in a number of steps. Instead of controlling the progress of an amorphous overall process the individual steps can now be validated. Instead of stating "Yes, we are working on the program and I think we are making progress" the project leader can now declare "We have finished phase 3 of this project in time".

For large projects that lasts many years or for projects where many people are involved, a well defined project structure and progress control is more important then for small pro-

[1]Actually we only consider the top level of the software process in this chapter. A complete software process contain much more details. Examples of these `details' are the documents used in the various phases and the people and tools manipulating these documents.

grams. It is for instance obvious that the problems in computing the sum of the first 17 primes, developing an automated student database and the programming language Clean are quite different.

Computing the sum of the first seventeen prime numbers is a small and well defined task that can be done quickly by one person. There is no reason whatsoever to introduce the whole machinery of a well-structured software process.

The university database is much more complex. The definition of the problem to solve needs to be made more accurate in a requirements analysis phase. In further steps one needs to specify how these needs will be served, what tools will be used and how the program will be structured before the actual programming can start. After the program has been finished it must be tested, loaded by the current student data and introduced to the users. As soon as the database is in use, it enters a maintenance phase where errors are corrected and new possibilities are incorporated. Structuring the software process in a suitable sequence of phases and controlling the quality and schedule for these phases makes it better manageable.

For a project as the development of Clean it is impossible to define the requirements and the design of the complete system before the implementation starts. There are too many degrees of freedom to do this once and for all at the beginning of the project. In these situations we need to use a form of evolutionary development. First the best possible approximation is constructed using all currently available knowledge. The obtained system in improved in a number of iterations. The requirements of the next iteration can be formulated given the experience obtained with the current version. This is the major reason of existence of the various versions of many large software systems. Each iteration can be structured as a sequence of well-defined phases.

Even we you have decided to use a structured software process there is still much freedom in the number of phases and the amount of detail that is required in each phase. In general it is required that each phase produces one or more deliverables. These deliverables are documents in some language. The language to use can be for instance pictures, English, mathematics, some formal specification language or a programming language. Adding enough structure makes the project manageable and measurable. However enforcing to much structure makes it boring to write the documents at one level since they prescribe all choices of the next phase and removes all room for creativity. The trade off between creativity versus structure is important and difficult. Among other things it is influenced by the nature, size, budget and time schedule of the project and the skills of the people that are going to accomplish it. An untrained crew needs more guidance than a high skilled and experienced team.

1.2 Waterfall model

The most famous software process model is the waterfall model. In this model the software process is spilt in a sequence of consecutive phases. The output of each phase is the input for the next phase. The number of phases and their name varies in the literature and on the needs of a specific project. A full fledged waterfall model can consists of the following phases:

Feasibility Study. The task of this phase is to give a global and informal description of the problem to solve. The costs and benefits of the product to construct should be estimated and evaluated. This is the time to consider alternative solutions like buying an existing product. The conclusions can be recorded in a feasibility study document (FSD).

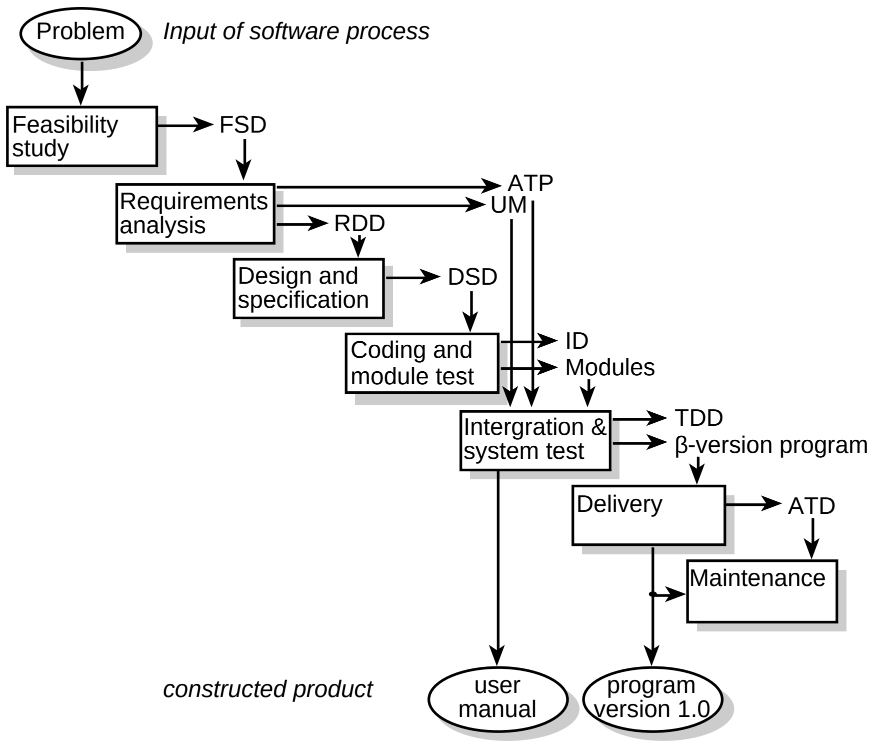

Figure 1.1 The waterfall model for software development.

Requirements analysis and definition. In this phase the qualities of the system to construct are determined. This should focus on *what* the system does, not *how* it does this. Among other things the following aspects can be distinguished: languages to be used, deliverables to be produced, functional specification (what the system should do), user interface, target platform, portability. This phase produces the requirements specification/definition document (RDD), an acceptance test plan (ATP) and a draft of the user manual (UM). The ATP prescribes how to test whether the constructed product meets the requirements. It can be necessary to develop a prototype implementation in this phase in order to validate the requirements. Since the prototype is a piece of software, it can have its own small waterfall.

Design and specification. Now it is time to design a method to solve the problem. At a high level the *how* corresponding to the *what* of the previous phase is determined. To do this the system architecture (the modules, their dependencies and their interface) are determined. Also the algorithms and datastructures to use are developed. These decisions are recorded in the design and specification document (DSD). Whenever it is not possible to determine the suitability of the design statically, a prototyepe can be used also in this phase.

Coding and module testing. The algorithms are written down in the selected implementation language. This is called programming in the small. The modules are tested on an individual basis, their co-operation is not yet analysed. Testing includes the correction of the errors spotted. Apart from the modules it can be necessary to produce an implementation document (ID).

Integration and system testing. The individual modules are integrated to one coherent system. This is also called programming in the large. This α-version of the system is tested. This phase produces the α-version of the system, the user manual and a technical test document (TTD) . The tests to do are described in the acceptance test plan created during requirements analysis.

Delivery. When the system is approved in the previous phase the β-version of the system is distributed under selected customers to see whether they are satisfied with it (β-testing). When the β-test succeeds the product is distributed among all customers in version 1.0. Results of this phase can be recorded in the acceptance test documentation (ATD).

Maintenance. A piece of software is hardly ever finished. Remaining bugs, new wishes of the user or a changing environment make adaptations necessary. Realisation of this changes is usually called maintenance. Usually changes are collected to make a new release of the system.

The waterfall model for developing software is depicted in figure 1.1.

Although each software project has to go through all these phases, the product of these phases is heavily dependent on the size of the project. For a tiny project a design in the mind of the programmer suffices. For larger projects an outline of the system architecture is required. For elaborate or complex projects documents of a serious volume have to be produced. From experience it is known that most software projects are underestimated. This implies that it is sooner worthwhile to put serious effort, including writing documents, in the preparation of the actual programming than you expect. In effect many software project come in serious problems or even fail just by lack on sufficient preparation and misunderstandings.

When a serious error is discovered during one phase the error has to be traced back upstream until its source. The project has to backtrack form that point. In general it is more expensive to repair an error when it has survived more steps in the waterfall. Due to the existence of errors the phases of the software production are less clearly separated than the management likes. A lucid separation of the various phases enable the management to monitor the performance of the project.

Not all phases of the waterfall model take equal amounts of time. Early data [e.g. Zelkowitz 78] show the 40-20-20 rule: on average 40% of the time is spent on preparation of the actual programming, the actual programming phase takes 20%, testing and error correction takes another 40%. In the waterfall model, preparation of the coding consist of the feasibility study, requirements analysis, specification and design. The distribution of time consumption of these phases is depicted in figure 1.2. Note that these figures only cover the time to deliver the first version of the product, maintenance is not included.

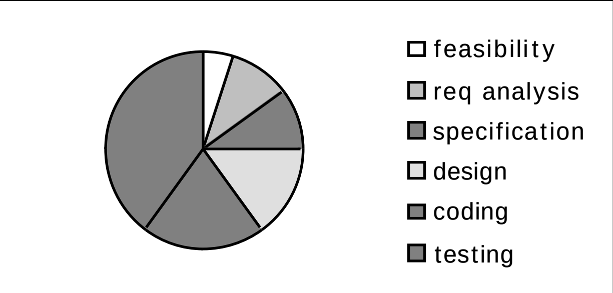

Figure 1.2: Typical time consumption of the phases to deliver the first version of a system.

More recent figures [Boehn 87] show an 60-15-25 distribution for successful projects. The larger amount of time spent on analysis and design is justified by the observation that it is far better to prevent errors than to try to repair the effects of these errors during testing. Especially errors made during early phases like requirements analysis and design are extremely expensive. A simple programming error may be hard to detect, but it is usually easy to fix. The additional effort put in the early phases causes a reduction of the total effort needed to produce a program. Other figures show that 50 to 75% of the total effort spent on a product is consumed by maintenance [Boehm 76, Lientz 80]. See figure 1.3 for the time distribution of the software project, obeying the 60-15-25 rule, including maintenance.

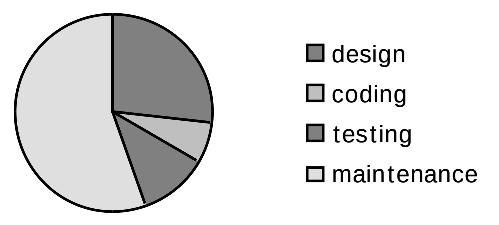

Figure 1.3: Typical time consumption of the phases of a complete successful software project.

The maintenance phase can be divided in a number of different activities. Note that the maintenance is not caused by wearing, but by the appearance of (hidden) defects and changing requirements. We distinguish four kind of maintenance activities [Lietz 80]. See figure 1.4.

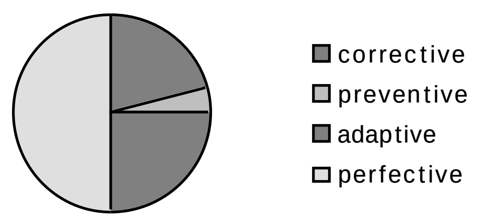

Figure 1.4: Distribution of maintenance activities.

Corrective maintenance, the repair of actual errors. This category has the best resemblance to maintenance in ordinary life[2]. This is a minority of about 21%.

Preventive maintenance increases the maintainability of the system in the future. This can be done for example by improving the modular structure or the documentation. This activity is relative small, only 4%, and usually done as preparation on another modification.

Adaptive maintenance has not the intention to correct errors, nor to improve the product. The only purpose is to let the current system run in a changing environment. Examples of such changes are new versions of the operating system, or a new version or a different brand of accompanying program like a database management system. These changes are not rare, they take about 25% of the effort spent in maintenance.

Perfective maintenance are all activities to improve the system. This improvement consists of adaptation to changed or entirely new requirements. This ranges from a better user interface to the same system to adding entirely new features. These activities form amount 50% of the maintenance activities.

1.3 Program development strategies

The early phases of the software process should provide a clear description of the product to construct and how it should be constructed. There are several ways to construct the program. Any serious program will be composed of several modules. The first step in

[2]When a hidden error in your car shows up, you will usually claim free repair based on your guaranty. For software this is still very rare.

the program construction will be to create a module structure including their relations. At this stage you will only have the module names and a global description of their task.

The next step is to define the interface of the modules by constructing the entire .dcl files. This phase includes the design of the major data types of the program and their manipulation functions. In this phase you also have to decide whether a data type will be exported, or that it will become an abstract data type and only its manipulation functions will be exported. The advantage of exporting the entire type is that this makes manipulation of the type in other modules easy and transparent. The disadvantages of exporting the type, and hence the advantage of making an abstract data type of it, is that the information of the type is spread all over the program which makes it much more difficult to change it later. When you want to enforce constraints on a type the could not be checked by the type system you are almost obliged to use an abstract datatype. Simple examples of such constraints are balanced trees and sorted lists. Manipulating this kind of structures as ordinary types is a source of errors that are hard to detect and locate. However, when you have to provide an interface to the abstract type that enables each and every manipulation of the type, there is hardly any reason to make this type an abstract type.

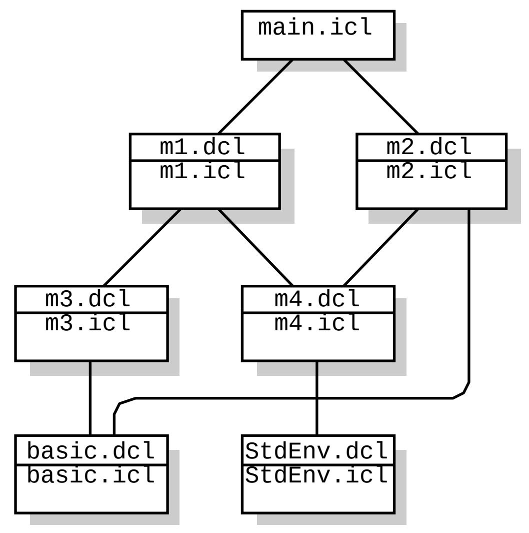

Figure 1.5: An example module structure.

Designing an appropriate module structure and module contents (datastructures and access functions) is a key step to a successful implementation. A weak design can hinder the rest of the implementation severely. Moreover, an unsatisfactory module structure is hard to correct and work around. Like usual in programming, there is no method that guarantees success in obtaining an appropriate module structure. The only thing you can do is to pay the desired attention to this phase and make an accurate design. The major guideline is that the modules must be well suited for the tasks of the program. A well designed module structure is usually easy to explain. During the rest of the implementation you will discover whether your choices were right or wrong. Do not hesitate to change your code when a mistake is detected. Many major implementation projects have had one or more major revisions, or ought to had them.

The development of appropriate datastructures received much attention in this book. The development of a module structure obtains much less consideration. This does not imply that there is not a module structure in these examples. The module structure is apparent is the implementations supplied with this book. For these examples the design of an appropriate datastructure was much more important than the module structure. In order to keep your attention focused on the datastructure and associated operations, the modules involved were ignored. As a rule of thumb, each coherent set of functions and corresponding datastructures should be put in a separate module. As an example each layer in the description of the machine architecture of chapter II.5 is a separate module. Moreover, you should separate general tools from their application by putting them in individual modules. By way of illustration, the parser combinators given in chapter II.4 should be in one module, their application in a concrete parser ought to be another module. This greatly contribute to the reusability of the constructed program fragments.

When the module structure is determined and their interfaces have been defined the corresponding implementation modules must be constructed. Although you have to provide an implementation for each module and the compiler is does not care in which order these modules are written, the ordering can be important for the programmer. The implementation modules can be written in any order, even a parallel or interleaved one. We will shortly discuss some extremes of this spectrum and mention some advantages and disadvantages of these strategies.

The program development strategies listed below can be used for the entire program, programming in the large, as well as for the individual parts, programming in the small.

Top-down

A well-known strategy is the top-down implementation of modules. We start with the implementation of the main module and gradually traverse the module graph down. Inside a module we begin with implementing the exported functions or the `Start` rule. We continue to define the functions in terms of simpler functions until they are all expressed in terms of basic language constructs or imported functions.

The advantages of this methods is that it supports problem solving using the divide and concur pretty well. The problem is split into a number of simpler problems. Each of these problems is split again, until you can formulate the solution using basic language constructs. The division of problems is guided by the defined module structure.

The biggest disadvantage of this method that it is not possible to use the compiler to check the modules you have written until the entire implementation is finished. This makes this strategy less suited for programming in the large, for programming is the small it is well suited and heavily used. The problems with using the compiling can be circumvated partly by simplified versions of the low level functions and or modules to enable testing of the high level stuff.

Bottom-up

In bottom up program development you start with the modules that are immediately built on the primitives of the language. The next layer is constructed on top of the constructed modules until the main module is constructed. In programming in the small you construct more and more complex functions and datatypes until the exported functions can be expressed in the imported stuff.

The advantage of this strategy is that the compiler can be used to check and test each module, even each datatype and function, as soon as it is constructed. When the main module is finally defined all other modules are approved correct by the compiler and should have been tested thoroughly. This makes testing and debugging of the main module much easier. The problem with this strategy is that many people find it difficult to define a hierarchy of increasingly powerful tools, without actually knowing whether they are suited to solve the problem. Again, the module structure defined is an important guide for the construction of the entire program.

An example of bottom-up construction is the machine description in part II of this book. We started with defining the components of the state of the machine and added a sequence of extensions one by one until we reached a third level language.

Incremental

Both strategies discussed above hamper us to get a version of the product under construction before the entire program is finished. Having an idea of the program under construction is important to convince you that you are working in the right direction. In order to support this, a program can be constructed incremental. The purpose is to get a simple version of the program running as soon as possible. This simple version is extended to the full program step by step. These partial programs are used to verify that we are constructing the right program. Moreover, most people find it stimulating to have a program running in an early stage and to watch the growing of its capabilities. Testing the program after each extension makes debugging less difficult. When you have several possibilities for an increment and select the best one empirically, this approach is also called exploratory programming.

In order to achieve a simple version of the program running of the program you have to cut of many pieces with courage. The pieces you want to include in the first version depends heavily of the kind of program you are constructing.

For the constructing of a drawing application the user interface will be probably of great importance. Here it is a good idea to start with the definition of the menu structure and windows. All event handlers will do nothing at all and input/output to files is not available. As first extension you will add simple drawing possibilities. In successive step other drawing tools, grouping, cut and paste, file io etcetera is added.

For programs where the interface to the user is not of prime interest an advanced interface can be constructed in a much later stadium. For the time being we can work with a simple line based interface instead of a sophisticated window interface. Examples of this kind programs are the parsers, machine descriptions and database management system discussed in part II of this book.

The waterfall model for software development does not fit well for this development strategy. Due to the huge number of versions it is impossible and unnecessary to go through all phases and write all documents for each version. Since the progress can be deduced from the increasing capabilities of the various versions, it is not needed to have all these documents to measure the progression of the project. Nevertheless it is worthwhile to investigate as good as possible what requirements for the final product are, to plan a module structure and to record the major implementation decisions. Below we will intriduce the spiral model for software development which fits incremental development much better.

Evolutionary

Evolutionary development is used when the product to construct cannot be defined entirely before the implementation starts. The early versions are not just used to convince us of the suitability of the design, but are also necessary to determine the extensions. In incremental construction you will have a clear description of the system to construct. It is built in a number of phases in order to facilitate testing and to validate the constructed system in an early phase.

In practise this difference can be less clear. Its is possible to postpone the decisions about some details of the design until we have some experience with the first prototype or to adapt the design slightly based on the knowledge obtained from the earlier versions. Creating a continuous scale from incremental to evolutionary development.

1.4 Spiral model

As argued in the previous sections the waterfall model for software development is not very suited for incremental production of software. As indicated in figure 1.3 above, in most software projects more effort is spent in the maintenance phase than in all other phases together. This indicates that the waterfall model may be suited for the construction of the first version of a software system, but not for the total life cycle of the software. We want to have some form of confidence that the effort spent in the design phase is not wasted. The best way to prevent errors in the design phase is to construct prototypes. In section I.1 we mentioned already that also the construction of a prototype is a software project that deserves its own life cycle. These observations caused the development of several process models as alternative for the waterfall model. The essence of the models is that phases similar to the waterfall model are visited again and again. We obtain a closed loop instead of the waterfall. To show the progress and to distinguish various iterations a spiral is usually preferred instead of a loop. Such a software process model can be depicted as shown in figure 1.6. See [Boehm 88] for a similar spiral model for software development.

Passing all phases of the spiral model once is called a cycle or a loop. The differences with the spiral model of Boehm are the number and names of the phases in a cycle and the fact that Boehm's model prescribes a fixed number of cycles. Each of these cycles has a specific purpose: determination of the concept of operation; requirements validation, design validation and verification, and finally implementation of the product. In our model each cycle prescribes just one evolution step of the product.

The number of iterations is not fixed, but depends on the project at hand. For a simple project, where an old fashioned waterfall model suffices, a single loop is adequate. The feasibility study corresponds to the review phase. Determining the goals was called requirements analysis in the waterfall model. The design phase was formerly called design and specification. Implementation corresponds to coding and module testing. The evaluation phase seems new, but contains the work previously done in the phases integration and system test and the testing of the delivery phase. Maintenance is modelled as one or more new loops of the spiral. In fact the current version of the software is just the prototype of the next version.

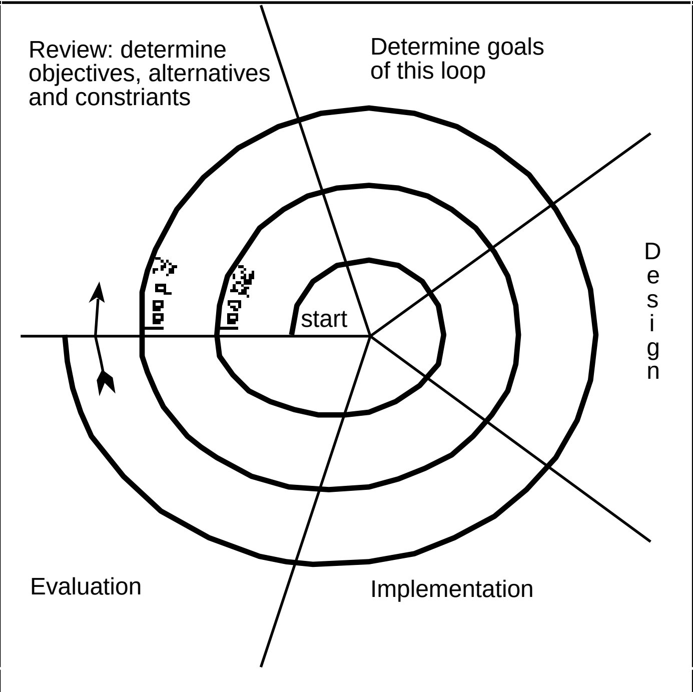
Review: determine objectives, alternatives and constriants
Determine goals of this loop
Design
start
Evaluation
Implementation

Figure 1.6: The spiral model for software development.

When a prototype is used the spiral has two loops. In the first loop the prototype is developed, in the next phase the final product is constructed. The final loop corresponds to the traditional waterfall model, and people used to this model are willing to split the implementation phase in parts corresponding to the phases found in the waterfall model.

For evolutionary or incremental development there are many loops in the spiral model. In the review phase of each loop it is determined whether it makes sense to continue this project or not. When you decide to continue you have to decide which extension of the project has to be made in this cycle of the spiral. Although it might be tempting to do 'the easy parts' first, in order to make a lot of progress fast, this is not a very good strategy. It is much better to do the parts that cause the highest risk first. Risk is here any factor that can obstruct the success of the project. Risks can be very different in nature. Examples of risks are: the availability of a good implementation of the programming language used (no running programs without proper language implementation), a thorough understanding of the goals of the project (are we making the desired product), the possibilities to develop an appropriate algorithm (no program can be made when we do not no how to solve the problem), and the availability of the needed data (the algorithm cannot be used when the inputs are not available). The rationale after the rule to select the highest risk task first is to limit the investments in a product that is doomed to fail anyway. A project that will fail in the end can better fail as soon as possible. So, it is better to determine whether the project will succeed or fail as soon as possible by performing the task with the highest risk first.

Notice that the 40-20-40 and 60-15-25 rule are only valid in the traditional waterfall model. More recent numbers [Zameroni 95] show that about 40% of the time is spent in coding, both for a sequential and for an incremental model. The incremental model requires a bit more time in the design phase, since evolutionary prototyping requires a better-designed system architecture. On the other hand, α- and β-testing of the incrementally constructed product requires less time. The total production time of a system developed using a sequential and an incremental model is very similar. The big advantage of the incremental product is the reduced risk. While the risks of the total project remain high until the β-test in the sequential model, it decreases by every prototype constructed in the incremental model.

An additional advantage of the sprial model is that it prevents the "big-bang effect". In the waterfal model it is quiet for the user between the requirements analysis and delivery of the β-version of the system. In the spiral model the user sees the system growing, can guide the development of the system, and gets used to the system before it is fully operational.

The decisions and products off each phase and each loop must be documented appropriately. Although documentation is extremely important, the rules for the documents to produce are less strict than in the waterfall model. Even for small projects it is very useful to write down the decisions made. Very soon you will not be able to remember what excellent reasons the were for many important decisions. After some time you will be forced to reconsider whether the decision made was the right one seen in the new situation of the project. This is very hard when the rationale behind the original decision has disappeared.

In contrast to what figure I.6 suggests, it is not always possible to draw a sharp border between the various phases in one cycle of the spiral. In fact the review of the current iteration is closely related with the determination of the goals for the next iteration, it would be reticules to determine goals without having a faint idea about their realisation etcetera. It is not necessary that all phases take equal amounts of time. Empirical data [Zamperoni 95] shows that the fraction of the time spent on analyses decreases with the number of the iteration. The fraction of the time spent on realisation increases each cycle.

The Cleanroom approach [Mills 87, Selby 87] to software development does separate the phases in the incremenral model firmely. The software is constructed according to a formal specification. After the syntax check the code is not tested, but proven to be correct. Testing is done only at the integration level.

The double helix model

When a functional language is used, the same formalism can be used during design and implementation. This enables the construction of a new version of the program corresponding to one cycle in the spiral in an incremental way. This can be viewed as incremental programming in the small. When we decide that the next version of the program under construction should have extensions *x*, *y* and *z*. We add these extensions one by one. For each of these extension, we design a way to construct, implement, and test it. This incremental in the small way of programming is very convenient since it gives immediately feedback. This feedback can give great satisfaction and works very motivating.

A prerequisite for developing software in this way is to have an efficient compiler. The joy of seeing the program grow is eliminated by the frustration of waiting for the compiler over and over again. Fortunately, the Clean system contains a very efficient compiler [Hartel 95].

A double helix is an even more appropriate model for the construction of software using incremental development in the large and in the small. The main helix is corresponds to the spiral as outlined above. The loops of the second helix corresponds to each incremental extension in the small. The second helix consists of the phases design, implementation and evaluation. It replaces the implementation phase and large parts of the design and evaluation phases in single spiral model. What remains in the design phase of the mail spiral are the global extensions or modifications needed. The evaluation phase of the main spiral consists of checking whether the goals of this phase are achieved.

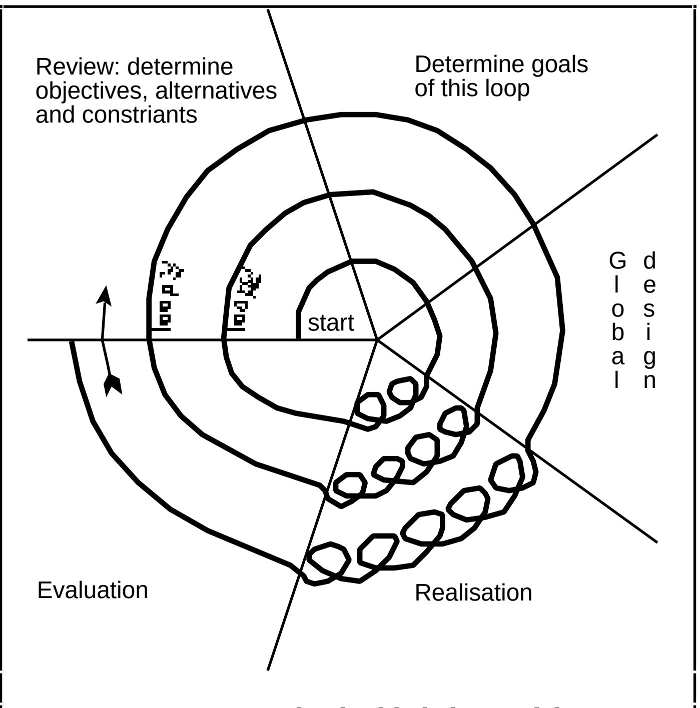
Review: determine objectives, alternatives and constriants
Determine goals of this loop
Global design
start
Evaluation
Realisation

Figure 1.4: The double helix model.

The versions produced by the iterations of the main loop are the official prototypes that can be used for instance to monitor project progress and/or to get feedback from the users. The huge amount of versions corresponding to the incremental development in the small are less accurately planned and exists only for the program developers. The essence of the double helix model is that there are periods of evolutionary development with small increments of the product relieved with periods of global reflection. The rationale behind doing the high risk tasks first carries over immediately from the spiral model.

A software process models should not obstruct the effective and creative construction of a product. On the contrary, the effective production of software should be supported by an appropriate software process. In our experience software production according to the double helix model is very effective. Most of the changes are small and the correctness and suitability of these changes is checked immediately. From time to time we step back to investigate whether the overall development of the project is satisfactory and which aspects must be developed with priority. The software process is not a goal on its own, but should be used as a guide in the construction of high quality software.

We should not hesitate to correct mistakes made in previous phases whenever they are discovered, even when the software process does not prescribe this. In a software process we can prescribe testing and reflection. How to react on wrong or bad decisions, however cannot be specified in the software process. The reason is that human beings appear to be marvellously creative in making errors. The impact of these errors varies from negligible to a tremendous. Some errors should be fixed before any progress can be made, while the correction of others can be safely delayed.

1.5 Functional programming languages

Using a functional programming language like Clean in a project has several advantages. The impact is much greater then the 20% (see figure 1.3) or 40% (see section 1.4) of the time spent on actual coding. Due to the mathematical nature functional programming languages are very well suited to express designs. The advantages above ordinary mathematics are the ability to use the compiler to check the consistency of the specification: the compiler verifies the correctness partially by type checking and indicating unbound identifiers. Moreover, the specification *is* the prototype implementation.

In chapter II.5 we saw for instance how the semantics of a simple third generation language can be expressed in Clean. This definition is at least equally clear as a conventional semantically description. The Clean compiler can check the partial correctness of such an specification by verifying type consistency and the proper definition of all identifiers. The absence of pointers prevents dangling pointers. Having no variables eliminates problems with initialisation and unexpected updates. People used to imperative programming are often amazed to see programs work correctly as soon as they compile without errors. There is no guarantee that a program that compiles without errors will work correctly, but it is a nice experience shared by many programmers in a lot of examples.

The quality of a proposed user interface can be judged much better from a prototype implementation than from a set of sketches of windows and dialogues on paper. In such a prototype the event handlers are do-nothing functions or simple toy functions approximating the final functionality. A prototype is superior when it is compared to sketches on paper since it can show the dynamic behaviour: the look and feel of the final product.

Since modern functional programming languages can be used to write very efficient programs, no shift of paradigm is needed to turn the final prototype into an efficient implementation. In fact there is no border between specification and implementation. A prototype can be extended or optimized at a one datatype or function at a time basis. By using a modern functional programming language the program remains compact and comprehensible which facilitates maintainability and reuse tremendously. Also *referential transparency*, the fact that the value of a function application depends only on the arguments, contributes to the ease to write correct programs as well as to maintain and reuse them.

Using a functional programming language enables you to use the same formalism for design, prototype construction and final product implementation. The strong type system performs consistency checks that otherwise would be necessary during testing. A part of the testing activities are performed by the compiler during the implementation. This is clearly a much larger impact then only the programming phase of the software project. Using an efficient compiler, like Clean, offers the possibility to work incrementally with very small increments.

Remember that there is no magic in using a functional programming language. It offers you just the possibility to things right, fast and comprehensively. However, it does not guarantee anything. In fact it is, even using high quality tools like functional languages, easier to spoil your project then to make it a success.

1.6 Software quality

Software quality has many aspects. In the waterfall model the quality of the software was mainly determined and improved during testing and debugging. Optionaly a prototype is constructed to validate the requirements analysis. A high quality product must meet the user requirements, should be well designed and properly implemented. This implies that all phases of the waterfall project must be done succesfully. The spiral models contain the same activities, but prescribe in addition the validation of a number of prototypes. Validation of these prototypes signals the construction of the wrong product or a mall functioning product.

Among other things, software quality is determined by the absence of errors. An error is a human activity that results in constructing software containing a fault [IEEE 83]. A fault is a manifestation of an error. If encountered, a fault may result in a failure. There is not necessarily a 1-1 relation between faults and failures, nor between faults and errors. A fault can cause different failures, and a failure can be caused by more than one fault. Testing is looking for failures. During testing we can only show the presence of faults, it is generally impossible to show the absence of faults by testing.

People have different notions of what a failure is. A very simple example of a failure is an unexpected abort of the program. In a functional program such an error can be caused by a missing case in the patterns of a function. This aspect of software quality is called *stability*. Furthermore, a programmer can consider each deviation from the specification or the user manual as a failure. The process of checking whether the program meets its specification is called *verification*. A user considers also deviations from the intended behaviour of a program as a failure. Investigating whether the program meets the expectations of the user is called *validation*. Finally, the project manager can also think of ill documented code or code that is hard to change or reuse as an error.

A bug is a small programming error. Debugging is the correction of bugs. Since the nature of the fault is often hard to predict from the observed failure, all error correcting activeties together are often called debugging. Even if the "bug" appears to have the size of an elephant. Moreover, debugging sounds better then correcting the large mistakes made suring requirements analysis. Some errors have tremendous consequences for the program. In worst case, when you have understood the customer completely wrong, the entire program can be worthless. Using evolutionary development, with regular feedback from the customer, such errors will be detected soon. In an evolutionary software process it is impossible to construct the wrong product accidentally. During the validation of the first prototype the user will indicate that this was not what he meant.

It is important to realise that running the program and looking for failures is an important and indispensable form of testing, but not the only way to find errors. Long before the first version of the program can be run there should be documents that describe the requirements and the proposed solution. The quality of these documents can be determined by a *peer review*. Also for the program a human inspection of the code can be beneficial. A peer can find errors for strange inputs (arguments) that are hard the detect during testing. Moreover, a peer can indicate problems with (implicit) assumption or maintainability that cannot be found by running tests.

An other important technique to detect errors is by constructing a *mathematical proof*. Incorrect programs cannot be proven correct, and hence all errors should be detected during the construction of the proof. The main reason why this technique is not widely applied is that it is a tremendous amount of work to prove programs correct. Additional problems are that a program can only be proven correct with respect to some formal specification. Testing can build confidence that the specification is valid, proving the correctness of the program with respect to the specification can't achieve this. Finally, there is the problem that the proof can contain mistakes. For instance, when you omit the same instance of a case analysis in the program and the proof everything looks fine, but is still erroneous. This is not only a theoretical possibility, but it is happened in published proofs of algorithms. Since proof does not give absolute security, nor validate the specification, they can't replace testing altogether. Since functional programs can be manipulated by equational reasoning formal proofs are far less cumbersome than proofs of imperative programs. We illustrate this in the section "reasoning about programs" below.

In the previous paragraph we discussed the possibilities and problems to prove an algorithm correct with respect to some formal specification. It is also possible to work in the other direction: *derive a program* from the formal specification. In a sequence of small and formally justified steps the formal specification is transformed to an efficient program. In this approach the proof and the program are constructed in conjunction. Programming and proving the program are not separated activities, but fully integrated. Due to the mathematical nature of functional programming languages, they are very well suited to derive programs from specifications. Since it takes usually more time to derive a program than to construct a program in the traditional way, deriving programs is not widely used. Derivations does not make testing superfluous for the same reasons as outlined above for proofs. However, you should consider that the derivation or proof replaces a part of the, often boring, test activities. Derivations of correctness proofs for commercial programs are usually limited to situations where extreme reliability is mandatory.

The problem with constructing high quality products is that there is not a method or activity that guarantees the desired quality. You ought to do everything right in the end and all errors that are inevitable made should be corrected. However, it surely helps if you always look at the work you are doing with the question *what can go wrong here*, and subsequently remove the possibilities.

Software maturity

The software maturity framework [Humphrey 88, 89] offers a means to assess the software process. The maturity levels themselves does not primarily concern the quality of the product directly, but the quality of the production process of the software. The basic assumption here is that a well controlled process is essential to guarantee a high quality product. There are five different maturity levels: *initial*, *repeatable*, *defined*, *managed* and *optimizing*. We will discuss the main characteristics of these levels briefly.

Initial. Software development is ad hoc. The organisation operates without clear procedures, project plans and cost estimates. This causes that it is easy to oversee or forget problems, maintenance causes real problems. The ability to deliver a high quality product in time in mainly determined by luck. In fact this is how we all made our first programming exercises.

A *repeatable* software process is much better under control. It is not just luck that the right product will be finished in time. The organisation has done it before and records change requests and commitments. Quality assurance guarantees validates the product. Through prior experience can yield similar results, control is required instead of educated guesses for this level of maturity.

At the *defined* level an organisation is not only monitoring the input and output of the software process (as in the repeatable level), but also the way how software can be developed successfully is defined.

At the *managed* level of maturity not only a successful way to develop software is defined, but in addition the software process is monitored quantitatively. Meeting the high quality goals is assessed through quantitative measurements.

At the final, *optimizing*, level of maturity improvement of the product is not longer the primary issue. Constructing a high quality product is completely under control. The organisation pays attention on how this product is constructed in the optimal way.

Humphrey also investigated the state of art in software production with respect to this framework [Humphrey 89]. According to his findings most software is was constructed at the initial level! There were only a few organisations operating at the repeatable level. The defined level was reached by a couple of projects. The managed and optimizing levels were unoccupied.

Currently many software companies have received an ISO-certification or are working to achieve this. The international organisation for standardisation, ISO, has produced several system standards for quality [ISO 87]. ISO 9001 is the model for quality that fits best for software engineering. ISO 9004 contains useful guidelines for elements of the various standards. ISO 9001 only covers the organisational aspects. So, this is not an alternative for technical requirements, but complementary to it. ISO 9001 does not guarantee quality, it only prescribes an organsition that can be targeted to quality. The topics to be addressed in an ISO 9001quality system are shown in table 1.1. The goals of the ISO quality standards and increasing levels of software maturity are very similar. We can expect that an ISO 9001 certified software producer operates at the maturity level called *managed*.

1	management responsibility
2	quality system
3	contract review
4	design control
5	document control
6	purchasing
7	purchaser supplied control
8	product identification and tracebility
9	process control
10	inspection and testing
11	inspection, measuring and test equipment
12	inspection and test status
13	control of nonconforming products
14	corrective action
15	handling, storage, packing and delivery
16	quality records
17	internal quality audits
18	training
19	servicing
20	Statistical techniques

Table 1.1: Topics of ISO 9001

1.7 Guidelines for software construction

As conclusion of this chapter we advise the following rules for the construction of functional programs:

0) *Make quality control an integrated part of the way you work.* This implies that a stability study, verification and validation are not just task that ought to be done at some time. These tasks are essential for the quality of the product to construct and should be part of all activities.

1) *Think before you start programming.* Make a thorough study of the product to construct, the users and the future of the product. Which existing products can be used. Which parts of the product to construct should be prepared for reuse.

2) *Think while you are programming.* This usually implies incremental development of the program. Be sure to pass the main loop of the double helix model several times. This means causes some reflection of the progress and the direction of the entire project regularly: *keep in touch with the purpose of you mission.* Testing of the increments should be an integrated part of your work. You should verify and validate the evolution of the program as a whole regularly. Be sure to record all design decisions and to document the code.

3) *Be prepared to changes.* During the incremental development it is likely that the much of the code you are writing will be changed. Prepare the code for these changes. among other things this means: use meaningful names, write down assumption you made. Consider what will happen when the function is used is with arguments that should not occur. Whenever possible, have these assumptions checked by the code you are writing. Even after you have "finished" the project it likely that the program will be changed during maintenance. Keep track of the cause of changes and their impact.

4) *Do not hesitate to replace parts of your program that are out of date.* Until recently people advocated to replace the prototype by a program constructed from scratch. This was argued by the difference in purpose of the prototype and the product: a prototype was seen as a vehicle to invest the requirements and to explore the possible solutions. The product should be robust and efficient. Using a language like Clean there is no reason to start all over again when the final prototype is to be replaced by the product.

5) *Don't forget to test the stability and suitability of the product.* Testing the increments separately is not sufficient to determine the overall quality of the product.