Fundamentals of Software Development

Module 2: Pre-coding Development

In this module, we examine our pre-coding development to look at problem-solving techniques, the specifications and then program design tools. Problem-solving is what the Product Owner does to determine the scope of the program being developed. Specifications are critical to program development, as they say what the program should look like to the user when running, as well as what the program does (and how it does it). Program Design tools are akin to an architectural blueprint and are used by the coders. Instead of being bogged down in actual program code in a programming language, we use these tools to develop the program "in English" or using some form of diagram. From there, it is much easier to build the actual program.

Part A

Problem-Solving Tools

Now that we've taken a look at the Software Development Process, let's look at the first box and how we approach the problem initially to solve it. There are several tools which we can use to dissect the problem. Some people are better than others at solving problems, but if you work at it hard, then you can always improve your problem-solving skills.

I should note here that problem-solving tools are not formalized tools, but approaches to understanding the problem at hand. After we undergo a thorough round of problem solving, we will then progress to specification development and formalized design.

The techniques to use for problem solving include:

- Asking questions
- Divide-and-conquer approach
- Emulating the manual or current method
- The building block approach
- The linear approach

Asking Questions

This is one of the best methods that you can use to gain an initial understanding of the problem. The questions that you should ask can start out simply, and then progress to more pointed questions. If you are ever at a loss in deciding which questions to ask, revisit your basic adverbs: who, what, where, when, how, how much? And you may do this in any order.

Who? Who will be the main user of this program? Who will use this program occasionally? These are important questions because they determine the audience of the program. For example, I once developed a system for a client that marketed veal (a meat distributor). The owner of the company was 70 years old, and his wife -- who acted as accountant -- was 71 years old. They had never used a computer in their lives. When asking this question, I discovered that neither person was planning to use the new system. The owner focused on marketing and relations with customers, while the wife was looking forward to retirement. The main people who were to use the program included the son of the plant manager (he was a salesman), and he was an avid computer user. Also, the office manager was to use the computer. And she was an experienced computer user. This meant that -- while it was important that the program be easy-to-use -- we could assume that novice users would not use this program, freeing us from worrying about elementary details and excessive hand holding in the program. Both of the main users would use the program everyday throughout the day. Because of this, we determined that the program must be able to handle at least two users simultaneously.

What? What do you want the program to accomplish? This is a central question. And it's amazing how follow-up questions will unearth many interesting desires of clients. Clients often feel that a new program will solve all of their problems, when in fact the program may only address a few problems and may exacerbate other problems. For example, I used to work for the Rand Corporation. We had a department called Engineering Services. The manager of Engineering Services constantly complained that he needed a bar code inventory system to keep track of all the parts that came in and left his department. His major complaint was that people who were not in the department would visit the department, ask for parts, and be given the parts by his employees. And this led to a lack of accountability. When we asked him this question, and then probed deeper, we discovered that bar coding all of his parts and creating and installing an expensive tracking system would not address his major complaint. It was quite feasible that his employees write-down the number or description of each part that they handed out. A pen and well-designed form are a lot cheaper than a bar code tracking system with laser scanner guns. His problem was actually all about access. Our solution? We replaced the open door to his department with a Dutch door and locked the bottom part of the door so that people could not enter his facility.

Why? Why do we want to automate this process? I believe that there should only be three valid answers to this question. One should automate a process only when it reduces costs, increases revenues, or increases productivity. If it doesn't do one or more of these things, then why bother? Of course, one could answer that the process might be more fun to perform. But that is really under the heading of productivity, isn't it? If you can't answer the why question with one of these answers, then it is not automation that you should be looking at.

Where? Where are you planning to use this program? Will it be in the back office? Or will it be at the front desk? Will someone at the loading dock be using the program? Will someone need to use the program from home? Will the program be used by outbound salespeople? While this question probably affects the hardware design of the overall system (networking, laptop views), it is nevertheless important in determining how the program should be designed. For instance, an accounting system can be designed to to handle order entry from deskbound operators, or it can handle orders taken by outbound salespeople via laptop computer. The design would be much different if the latter were the case.

When? When will this program be run? Will it be used throughout the day? Or will processing happen only at night? The answer to these questions will affect the programs interactivity. For example, take a bank. Most banks have programs that run throughout the day. However, the program that updates the account balances only runs at nighttime. The daytime programs are highly interactive. The nighttime program however is a batch program that has little interaction with the user once it has been started.

How? How do we approach the development of this program? The client may wish to phase in the use of this program. Or, perhaps there is a problem that needs immediate correction, and the program should be deployed as soon as possible. Not only does this effect your development schedule, but it might also affect how you build the program. In a rush situation, you may choose to use prefabricated modules.

How much? How much are you willing to spend? That is one of my favorite questions to ask clients. Often, they don't have an answer. Most of the time they have an answer, but they really don't want to tell you. Through careful salesmanship, you can figure out the range of money they want to spend. Simply put, the more money they have to spend, the more features you can put in the program. Or the more solid you can make the program. Or the more customized you can make the program. Hopefully, a combination of the three. If they have little money to spend, maybe you should propose that they buy a program that is already created. There is usually a solution for every budget. Keep in mind, though, the Internal Revenue Service has spent millions of dollars trying to develop modernized programs to handle their needs. Hundreds of millions of dollars later, they have failed at every turn. So money isn't everything.

The Divide and Conquer Approach

This is a classic approach to solving problems of many types. Simply stated, take a problem. Break it down into easier problems. Now looking each of these easier problems, if it doesn't appear easy to solve, break it down further into even easier problems. Keep doing this until the entire problem consists of easy problems. Then you just have a lot of easy problems to solve. Look at your company's org (organization) chart. The president of an organization can't do everything. So he or she relies on vice presidents to handle each area. The vice presidents rely on various directors to handle sub-areas. Those directors rely on managers, and the managers rely on supervisors. The supervisors oversee people doing relatively small tasks and projects. One could say that the ultimate problem solver was Henry Ford in developing his Model A assembly line!

We will see that this approach is used in virtually all programs, and even evolves to one of our formalized design tools.

Emulating the Manual or Current Method

When in doubt, you can always rely on this method (whether it's best or not needs to be seen). With this method, you simply look at the way the client performs the process right now (either manually or automated). The manual method should give you some clue as to how to automate the process, but if it is already automated, the current process clues you in on possibly how to proceed anew.

Sometimes the current method is inefficient and the new automated method will greatly improve the efficiency. Other times the current method is highly efficient (even for a manual method), and the new automated method will simply make a good thing even better. When combined with the asking of questions, emulating the manual/current method gives you a very strong start.

Be careful, though. I've witnessed projects where the previous manual method was very limited, and the proposed automated method did not use the power of the computer to enhance efficiency.

I was told of an ill-fated simulation project back in the 1960s. To enter programs and data in the 1960s, people wrote the program statements and the data onto punch carts via a machine called a keypunch. One company wanted to replace their keypunch machines with new video terminals. They had a programmer write a program which would emulate the actions of a keypunch machine. But considering the drastic limitations of a keypunch, this was highly laughable. Imagine having a full-screen terminal and only being able to type a single line. If you needed to edit the line, you would press the reject button and re-type the line from the beginning, which is how you did things on a keypunch machine. Obviously, this was not a good implementation of automation.

Try to look toward the essential components of the process rather than the implementation. Those components will point you in the right direction.

The Building Block Approach

This approach is used in conjunction with the divide and conquer approach. After you have established the hierarchical chart associated with the divide and conquer approach, you find that one particular module in the chart could be satisfied by using existing code. This might be code that someone in your group wrote earlier, or it might be code that you purchase and include with your program. In essence, one is using prefabricated modules here to complete the program. Whereas in the 1960s this was not the ideal (everyone wanted to reinvent the wheel), now this is considered a critical element of program development. In fact, the object oriented programming movement gained significant popularity simply because its components can be reused in new programs. This saves time and money.

As an example, when I worked at Ashton-Tate, we released a product called Framework. Framework was one of the first integrated software packages for PCs. It included word processing, spreadsheets, database management, business graphics, and communications. During the initial development period, the major parts of Framework were being produced in timely manner. However, the communications module lagged behind. Soon the release date loomed over us. We had to make a business decision to purchase an existing communications program, and re-crafted a bit so that it appeared to be part of Framework. While this didn't necessarily save us money, it did allow us to ship a complete product at release date; it saved us a great deal of time.

The Linear Approach

This approach looks at the process in a time-based manner. If we look at the path we take through the process, it stops at several junction points where we have choices as to which route to take. Sometimes we have a choice of three routes. Other times, we do a task in the process in a cyclical manner and then once a certain condition is satisfied, then we move on to the next step. A good way to visualize this approach is to look at an airline system route map. The route map will show several ways than one can get from one city to another city. Likewise, we can do the same with a process that we plan to automate.

Let's take the example of a video rental store (are there any left?) and a DVD movie as it moves through various points in the store. While the DVDs all come from the store's regional distribution center, each individual serialized DVD title moves through a different path inside the store and throughout the rental process. An automation project would track the location of the DVD at all times. A DVD starts out by getting logged into to the store's inventory. From that point, it may be placed immediately on a display shelf or it may sit in the back waiting for a release date. If it takes the display shelf path, then it sits there waiting either for a customer to choose it or for a store employee to return it to the back. If a customer chooses it, he either takes it to the rental counter, returns it to the shelf after a change of mind, or places it elsewhere in the store (irresponsibly). If he takes it to the rental counter, it is either checked out or--if the rental somehow doesn't complete (perhaps the customer owes significant late fees and after paying them doesn't have enough money for this particular DVD)--it goes into a "go-back" bin for replacing on the shelf by a store employee. If the DVD goes home with the customer, then later it is either returned on-time, returned late (after a certain date), or is not returned at all (considered lost/stolen by the store after an even later date).

The DVD will go through many different routes during its life in that particular store. Our automation can keep track of the location and also provide all that possible routes and subsequent actions for each location. Using more technical language, we can say that at any given point in time, the DVD has a particular "state" with several possibilities for "transitions" or "outcomes". In theoretical computer science, students learn of a concept known as a finite state machine. The diagram for a finite state machine uses various bubbles and arrows to graphically indicate the various "states" and "transitions". Creating a finite state diagram is an excellent way to attack a problem using the linear approach. In fact, many disciplines (including economics and organizational behavior) employ charts which are very similar to the finite state diagram.

While the linear approach appears to be very different than the others discussed, it is really meant to be used in conjunction with the other approaches. Combining all of these approaches in your problem solving toolbox gives you an excellent weapon in dissecting problems to come up with an algorithm and eventually, specifications and a formal design.

Summary (Part A)

We can see from this discussion that there are many ways of looking at solving a problem. It is important that you are flexible, for each problem is different and will employ a different combination of methods. While your initial attempts at problem-solving will be challenging, with more experience the process will likely become easier. However, some people never quite get the hang of dissecting and attacking problems. If you are one of those people, don't despair; there are other employment avenues in information technology for you. They just don't pay as well. :-)

Part B

Specifications

Now that your problem-solving tools have given you greater insights on the problem to be solved -- and you have now created an algorithm, however rough -- it's time to formally map out specifications. Specifications are a form of documentation that describe -- in the clients language -- the program' s environments, the program' s basic functions (often with a high-level of detail), the user interface of the program, and a detailed description of all data files which will be input or created by the program, as well as output (written, spoken or otherwise). In a contract programming environment, specifications are critical. The specifications are usually referred to in the contract, and if they are not ultimately met by the program that you're going to develop, then your client will see you as having breached the contract. Specifications allow the clients to see exactly what they are getting before they get it.

By the way, the word "specifications" is a mouthful. Most software professionals use the word " specs". I will do so for the remainder of this lecture.

If I had to divide specs into two categories, I would divide them into general/functional specs and input/output/user interface specs.

General/Functional Specs

The general or functional specs are concerned with the basic actions that the program will perform, especially from a user perspective. The best way to view this is by example. Let's take accounting. In an accounting program (and in accounting in general), there are many rules that must be followed. These are known in the accounting trade as Generally Accepted Accounting Principles. They lay out what happens when the company sells a product and receives payment for its, or when a company needs to borrow money via a short-term liability. These principles would specify that for a sale transaction, a company's Sales account increases and so does the Accounts Receivable account. In accounting jargon, we would say that the Accounts Receivable account is debited, and the sales account is credited. When that company's customer pays their bill, Accounts Receivable is reduced by the amount of the payment and the checking account increases. Using jargon, we'd say that the Cash account is debited and Accounts Receivable is credited. So the Generally Accepted Accounting Principles lay out many rules to follow, even if you're in a completely manual environment. Our functional specifications would incorporate these rules.

Functional specs also describe environments in which the program is going to run. For our accounting program, for instance, information on the type of computer that is required to run the program would be given in the functional specs. Minimum processor, RAM, operating system version, necessary hard disk space: these are examples of the type of information we would need. If the accounting program is to run in a network environments, we would specify information on the network operating system, the number of users that will use the accounting program concurrently, and information on user security.

Input/Output and User Interface Specs

The input/output specs are often developed prior to the bulk of functional specs. The I/O specs are concerned with how the program is going to look when it's running and how it's going to interface with existing and new data files, as well as how it's going to output reports and other printed materials. If the program issues other types of output, such as spoken output or music, then these would be described in detail or examples given. Picture yourself as the client. You are getting ready to spend a lot of money on the development of a program. You already paid the development team a sum of money to think through the problem and come up with specs. Imagine your delight when the development team calls you into a meeting to show you what your program is going to look like. Using screen design tools, they can show you a mock-up of your program, including menus, dialog boxes, and other elements of the user interface. They can also show you sample reports and other types of output. And they can do this without the program actually being functional. In the '60s and '70s, this process was a lot more difficult than it is today. Today's Windows programmers are able to use visual environments to establish the user interface prior to worrying about functional elements. This is one of the reasons that Visual Basic became so popular so quickly, and database development environments such as Oracle and Microsoft Access are widely used. We've come along way from plain-jane print layout charts (like those shown in your text) in just a few short years.

Screen-oriented I/O specs are known as user interface specs. Many of the other specs are more technical in nature, involving data files, database files, communications, and other things that are described in very technical terms. While your program is probably going to create data files, there is a good chance that it will also read files that already exist. These files need to be described in detailed so that program can properly use them. As an example, say you were creating a new accounting program that is going to compete with QuickBooks. The user interface portion of the I/O specs would illustrates the screens used for invoice creation, check writing, register perusal, on-screen reports, customer summary screens, and dozens of other elements. It would also discuss how the keyboard and mouse are used in conjunction with the screens. The other I/O specs would discuss how you would get an existing QuickBooks data file into your new program, or some other type of imported data file. It would describe the format for the user data file, as well as the formats for any data files used by the program while in operation. It would describe the format for export files. It would also show the print layouts for every report that the program would produce.

What are User Stories?

When Scrum became a dominant software development methodology, it introduced a variety of colorful new terms into the lexagon. One of these terms is a new phrase for "specifications" -- "user stories". This term is also used by other agile development tools such as Extreme Programming.

The idea is that the design of an application should be driven by the user: what the app needs to do, how it should feel, what rules or laws or corporate procedures must it follow. In essence, a high-level description of the functionality of the program. But isn't this the same as the specifications I described up above? Yes, it is. It's simply a new way of expressing the design (and perhaps a less stuffy term and approach).

Here are some examples of user stories for a college registrar system:

- Students can purchase monthly parking passes online.
- Parking passes can be paid via credit cards.
- Parking passes can be paid via PayPal.
- Professors can input student marks.
- Students can obtain their current seminar schedule.
- Students can order official transcripts.
- Students can only enroll in seminars for which they have prerequisites.
- Transcripts will be available online via a standard browser.

Above is reprinted from agilemodeling.com.

User stories are often given a priority and an estimated size to eventually help the developers do their coding estimates.

Who Develops Specs (or User Stories)?

When viewing from the agile/Scrum perspective, user stories are developed by those who have a user-level, operational or financial stake in the resulting program (call these the "client" or the "stakeholders"), along with the Product Owner. The process classically separates these people from the developers in order to offer a clean user-driven perspective.

But in a more traditional setting, the development of specs should involve the systems analyst and development team as well as the client. Quite often, the client doesn't really have an idea how they want things to look. So it is incumbent on the systems analyst and development team to have good knowledge of user interface design. Many development teams have a person who specializes only in user interface design. This person is often not from a computer science background, but has a background in graphics, psychology, or human factors engineering. In the case of large commercial software firms, such as Microsoft or Netscape, focus groups may be employed to evaluate the user interface. Microsoft even goes further than this and tests new user interfaces in a usability lab. In this lab, technicians analyze ease of use of a particular interface by watching (through a two-way mirror) the test subject attempt to use the interface. The development team should also have strong knowledge of the functional area in which they are programming. For example, the development team for the accounting program should have at least one accountant on the team.

In any case, several iterations of specifications will generally result. And there is a lot of give and take between the parties involved. After many discussions and changes, the tug of war ceases and the client approves the specifications. Specs are usually written and presented in report format.

In a Scrum environment, the Product Owner will control and approve the user stories that go forward to the Scrum Master. While user stories are originally written on informal media such as index cards, the final version will problem be entered into a Scrum management system, similar to the one we saw in the Scrum video last week. They can then be moved around and assigned to the various backlogs easily.

Requirements

Some companies will use the term "requirements" instead of specifications. What is the difference?

It's best to think of specs as being the end product from which everything (e.g., code, documentation, test plans) will be generated. Requirements, on the other hand, are more of a detailed and required "wish list" generated by a client who then passes them along to development organizations for the bidding process. The developers' analyst teams will then spend time developing the specifications that satisfy these given requirements.

Requirements are not as detailed as specifications, as they are the framework from which specs are created. Requirements are generated from a variety of sources:

- User wants and needs
- Corporate business rules

- Industry standards (such as ISO 9000 Quality Assurance, GAAP)
- Regulatory compliance (such as Sarbannes-Oxley, Check 21)
- Federal and State Tax laws

In addition, for products that are generated by software companies for sale, their Marketing departments will conduct everything from surveys to focus groups to determine what the market wants in the product. This information will be distilled into a set of requirements, from which more specifications will be generated.

Examples of Requirements and Specifications

Both specs and requirements can be simple or very complex. There's no "typical". And in fact, for all of my explanations above about the differences between the two, the terms are STILL often used interchangeably.

- Here is a real-world set of what the author calls Software Requirements Specification.
- If our course was focused more on the detail of specs development (rather than just the basic concepts of it plus programming), we'd be spending several lectures on the topic. Here is a wonderfully detailed set of slides that outlines several examples of different types of requirements and specs (Links to an external site.).

Part C

Program Design Tools

Now that we have our specs -- and we have thought through what we need to do thoroughly -- it's time to design the program. However, we don't start writing code just yet. We need to map out what we plan to do first. Think of what happens when a building is constructed. Prior to actual construction, the architect goes through a problem-solving and specs process similar to what we have discussed. And then they must create blueprints from which to construct the building. Our formal program design tools create something similar to a blueprint. Once the formal design has been completed, a programmer can then code the program in the language of choice.

There are two classes of tools that will be using to design the program. The first class illustrates the overall organization of the program, while the second class is concerned with the logical flow of actions (the details). We'll use one tool from each class. It should be noted that this is not a sequential process. We do not design the overall organization, and then design the individual actions. We use the tools in tandem to design the program.

Program Organization

For structured programming, there's really only one tool that's part of the "overall organization" class -- the hierarchy chart (sometimes referred to as a module structure chart). The hierarchy chart looks quite similar to a corporate organization chart. Please refer to the sample hierarchy chart you can download from this folder. If we can picture the program as a group of modules, the hierarchy chart shows the relationship of the modules to each other. The top box represents the main program, which is similar to the CEO of an organization. As we move from the top of the chart toward the bottom of the chart, the modules become increasingly specific. In a corporate organization chart, the second level would consist of vice presidents, while the third level would consist of department heads. Moving down to the lowest level of a corporate organization chart gives us the non-managerial workers. They deal with specific highly tasks that--at first glance--don't even appear to be part of the overall effort. It's the same with the lowest levels of the hierarchy chart.

You'll recognize hierarchy charts from our discussion on problem solving. Remember the divide-and-conquer method? Well, this is the chart that illustrates it. Like the corporate organization chart, the hierarchy chart tells us nothing about the actual tasks performed by each box. It only shows overall structure. And in fact, the order of the boxes at a particular level is irrelevant. We cannot assume that--at the second level of the chart--the things represented by the first box happen before the things represented by the second box. It is not this chart's duty to give us that information. That is the task of the second class of tools.

Program Logic

The second class of tools illustrates program logic. Program logic is the term we use to describe what actually happens in each module. When we look at the tasks performed at this level, we have either a sequence of tasks, a loop (where some of the tasks are performed repeatedly), or selection (where at one point we have multiple paths to follow and the program has to choose one path over another). And I'd be more accurate if I said that each module consists of combinations of sequences, loops and selections. By the way, another term we use for program logic is "flow of control".

Although many tools exist for illustrating program logic, four tools are the most popular: flowcharts, Nassi-Shneiderman charts, pseudocode, and transition charts (or finite-state diagrams). On a particular project, you'll select which of these tools you wish to use. Each has its strengths and weaknesses.

Flowcharts are probably the most well-known of the flow-of-control design tools. They were very popular in the '50s and '60s. Using a series of shapes (such as rectangles, parallelograms, circles, ovals, and diamonds) along with several arrows, one can illustrate the program logic. Inside the shapes are words describing the action that's happening at that moment. And the shapes have meanings in and of themselves. For instance, a diamond means a decision (selection). And a parallelogram means input/output. Flowcharts are very easy for the casual observer to understand and follow. However, flowcharts can quickly become unwieldy. They can easily lack structure and this makes it difficult to code the program. They've largely fallen out of favor in the past two decades.

Pseudocode looks very much like the actual programming language. It consists of a series of statements that use universal symbols and English words to describe the actions. While it's important for computer science majors to learn the actual formalized pseudocode as described by ANSI or ISO, for you we feel it's as strenuous as learning an actual programming language. So when I ask you to do pseudocode, I'll loosen the requirements and let you use fully descriptive English sentences where applicable. Pseudocode tends to be more structured than flowcharts. Whereas the logic in flow charts can be difficult to follow, pseudocode is like reading a list of instructions. The only downside is that it is not immediately clear what type of program logic is occurring; one must always read the pseudocode statement to determine what's happening. This is only a minor annoyance in many situations. Pseudocode is quite popular in many circles. For the sample programs will be doing in the next week, we will use pseudocode.

Nassi-Shneiderman charts are probably the most popular of the tools used by educated programmers today for procedural programming. They combine the graphical simplicity of a flowchart with the straightforward nature and structure of pseudocode. The tool became popular with the rise in structured programming languages (like Pascal and PL/I) in the '70s. The Nassi-Shneiderman chart appears as a vertical rectangle. A sequence of actions appears as a stack of horizontal boxes within the rectangle. A sequence of actions that will be repeated in a loop appear to the right of an L-shaped trough. And selection appears as a series of columns. One can always glance quickly at a Nassi-Shneiderman chart and figure out the basic logic. There is one Nassi-Shneiderman chart for every box that appears in the hierarchy chart. Because of their structured top-to-bottom nature, they--by definition--cannot become unwieldy. We will use Nassi-Shneiderman charts a couple of times in the course. Here's a comparison of a flowchart (strongly discouraged today) and a Nassi-Shneiderman chart.

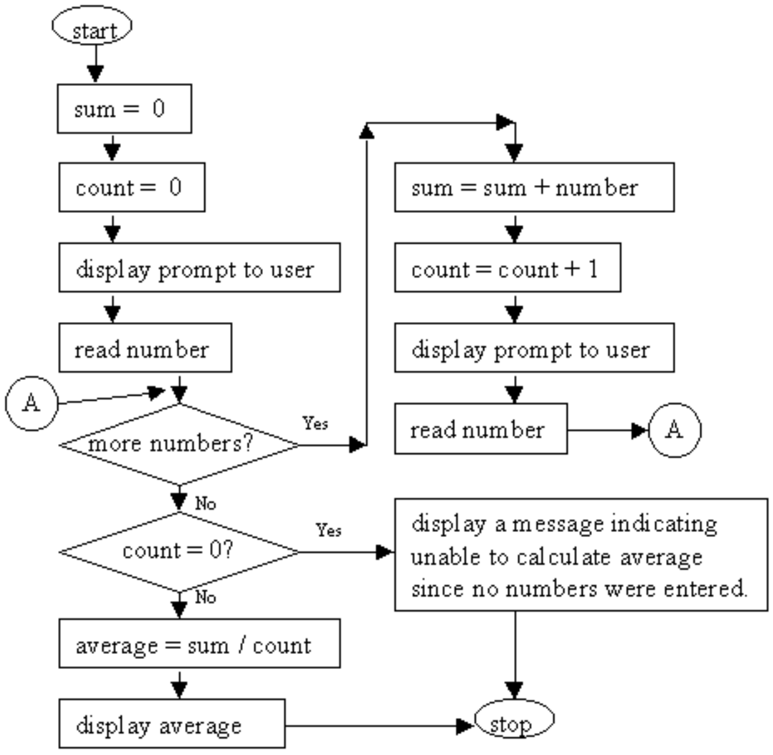

Figure 1. Flowchart

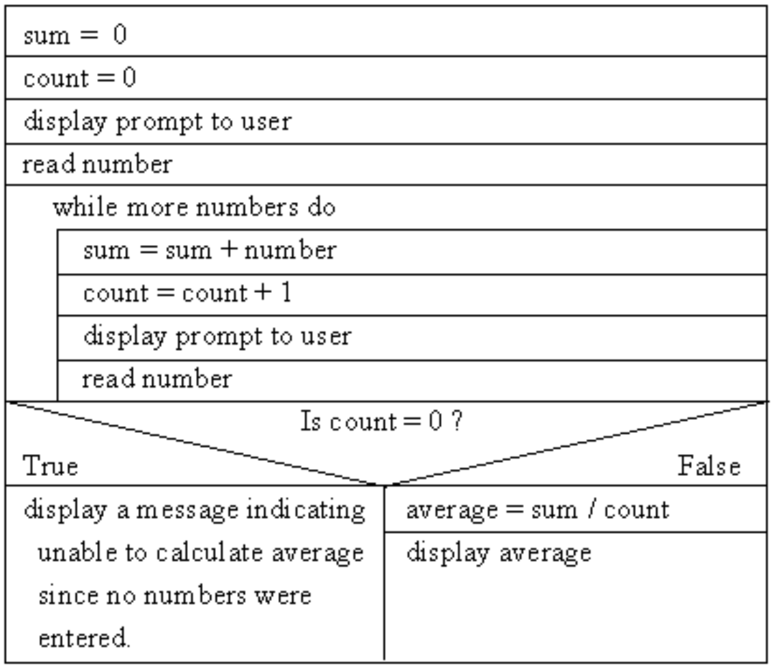

Figure 2. Nassi-Schneiderman chart

Transition charts and finite-state diagrams are most useful for today's chaotic non-procedural event-driven programs. Imagine running a Windows program. At any given moment, you can do one of many things: move the mouse, click it in the work area, click and drag it to select text or items, choose one of many menu items, right-click to get a context-sensitive menu. With event-driven programs, the user drives the program more than the program drives the user. Transition charts enumerate every possible action at any given moment and talk about where to go next. Finite-state diagrams show this in graphic detail. These charts look similar to airline routing maps (only worse). They utilize a series of "states" and arrows, with the arrows pointing from the current state to the "next" state, depending on the action chosen. Today's object-oriented / event-driven programmer has to use such a chart to get their bearings, possibly along with pseudocode or N-S charts. We will examine these diagrams more during our discussion of Visual Basic.

Using The Two-Types of Tools Together

Again, we cannot construct a hierarchy chart in its entirety prior to developing the Nassi-Shneiderman chart (or one of the other charts). They must be built together. We first draw the Main Program box at the top of the hierarchy chart. We then put our Nassi-Shneiderman hats on and look at the overall actions that are going to occur in the main program. We develop a single Nassi-Shneiderman chart to illustrate the actions in the main program. We then look at each of the actions to see if any are complex enough to be broken down further. We identify these with a name, and each of these names will go on in a box and that box will go in the next level of the hierarchy chart. So all the "hard" modules referred to in a single Nassi-Shneiderman chart will appear on the same level in a hierarchy chart. We then select a module to work with in hierarchy chart, and develop the Nassi-Shneiderman chart for that module. It's difficult elements are identified and those will become modules in their own right on yet another level of the hierarchy chart. Although modules can be developed by single programmer, what is common is development where the modules are parceled out to different programmers on the team.

Unified Modeling Language (UML)

Because of the complexities of modern-day programming (which can be object-oriented, database-driven, event-driven and all kinds of other adjectives), a new class of design tools was created. This set of tools--known collectively as *Unified Modeling Language* diagrams or *UML* diagrams is now widely accepted in the software development world. The set encompasses and refines many of the tools that had been in place for decades without standardization.

This web site presents examples of UML diagrams. This is for presentation only, not mastery. You will learn UML diagrams in your first object-oriented programming course, or in a separate UML course offered by UCLA Extension. Note that finite-state diagrams (called State Diagrams here) are a subset of the UML 2.0 specification.

After you complete this class and start taking database and object-oriented courses, it would be very wise to include a UML Design class in your curriculum along with your other classes. For this class with its simpler exercises, we'll stick with the simpler pair of tools described above.

Conclusion

Program design is a critical part of programming. You'll do it as part of each assignment, and I'll make sure that you do it well. During construction on it (and especially after construction is completed), you should test the design to make sure it works on paper. Start with the main program module and follow the steps outlined in Nassi-Shneiderman chart for that module. Then follow the design wherever it takes you. If it makes sense on paper, then it's time to develop your program code. If it does not make sense on paper, then go back to the drawing board (whether that means changing the design, or scrapping the design completely and attacking the problem from a different perspective).