Module 1: Software in a Nutshell

Here in our first module, we take a look at basic elements that are at the root of software development. First, we define some universal terminology. Then we look at various ways of approaching the software development process, from the classic method (from which many are derived) to one of the newer methods (such as Scrum) that reflect today's Agile & Rapid Application Development mentalities.

Part 1

An Overview of Software Development

You are about to embark on a course in software development (a fancier term for computer programming) with a business emphasis. In this course, you will learn many concepts involved in creating programs, and these concepts will be fundamental and necessary to move on into advanced programming. After this course, you will take courses in C#, Java or Python, database development with SQL, web design with HTML/CSS, PHP/MySQL, Adobe Dreamweaver and Animate, Joomla or Wordpress. This course gives you a foundation in programming that's necessary for pursuing any number of languages, products or approaches to software development. It should take you about 2 years to get through all of the courses in the certificate program you've selected. At that point, you will be ready to look for an entry-level programming position.

A few years ago, there were typically about 350,000 open programming positions in the U.S. at any given time. Because of commoditization of programming and offshoring, this number reduced drastically. But in the past five years, things have EXPLODED and there are now more than 500,000 coding positions! Most of them call for experienced programmers, but entry-level positions do exist in a variety of software development areas. Sometimes these positions are in traditional programming, while others are in the database arena, in web development or even in the rapidly growing worlds of both mobile development and game development.. If you keep going to school and studying while you work, you should expect salaries in the $45K-90K range within 3 years of leaving the certificate program. Different salary levels apply with regard to the area of software development you decide to enter. Experienced programming project leaders commonly see salaries in the $75K-150K range.

But you don't have to go corporate, or even to an existing small business. Many programmers today produce software products that are sold on the open market. Whether it is Apple's App Store, Microsoft's Windows Store or the Google Play Store, you can be a development shop of one and get your product out to the consumer marketplace. Additionally, you can develop extensions that can be used by web developers, especially those using platforms such as Wordpress and Joomla. These extensions significantly extend the functionality of these platforms. You won't be receiving a salary, but you could sell your product for $3 to 100,000 consumers a year, if not more. The software development world is FULL of opportunity.

Who Are the Players?

Up to this point, you have been the user of business programs. When you write a document using a word processor, or create a spreadsheet with Excel, or balance your checkbook with Quicken, you are considered a “user”. Even when you create macros for Word or Excel, you are still considered a user. A “programmer” is one who creates programs. A programmer is also known as a “programmer analyst” or “software engineer”. He or she develops a series of instructions to the computer to perform actions such as placing something on the screen, asking for input, reading from a file, or printing to the printer. Programmers work either alone or on a team (usually on a team), and work under a “team leader” or “project leader” or “lead programmer”.

Other people are involved in software development, including “systems analysts” (who look at the big picture of how programs, systems, and people direct the flow of data through a project or organization), “configuration managers” (who keep track of the various components of programs through development, including version control), “test engineers” (who test the program from a user perspective to report on bugs on the program), and “documentation specialists” (editors and writers who develop manuals, help systems, and tutorials for the program). In a commercial environment, you will also have a “product manager” involved. The product manager is a marketing executive who oversees every aspect of the program’s development as a commercial product to be sold, including features, schedules, markets, packaging, and support. You will now start viewing things from primarily a programmer perspective, although you always need to keep the user in mind.

Types of Programs

Programmers develop all the software you work with, including operating systems, driver files, business programs, networking software. These programs are divided into two main areas: applications software and systems software.

Applications Software

This software is anything that is applied to making a user do their job easier, cheaper, or more productive. Business software, scientific software, statistical software, graphics software, mobile apps and games are all example of applications software. To succeed in this area, it helps greatly if you have a strong knowledge of the application area itself. For instance, if you are developing an accounting program for a food distributor, you should know about accounting principles as well as food distribution practices. And if you are doing a mobile app that deals with geo-location, it's great to have a background in geography and mapping.

Systems Software

This software is what makes the computers run, and also allows for the creation and execution of applications software programs. The category includes: operating systems, compilers (which change your programs into a form the computer’s microprocessor can understand), drivers (programs that connect hardware peripherals to the operating system), linkers, loaders, print spoolers, bootstraps, and a variety of other programs. This type of program is considered more difficult than applications software, and consequently, systems programmers are generally paid more. They also tend to have degrees in Computer Science. If you think about the complexity of Microsoft Windows, the Android OS, Apple's OS X and iOS, or the software that runs your Canon multifunction printer/scanner, you can see why a CS degree is necessary for these areas.

How Are They Created?

Programming Languages

A program is created by writing instructions to the computer in a programming language. Many programming languages exist, but the most popular ones today (evidenced by software that was developed in them) include C, C++, Visual C#, Visual Basic, SQL, Oracle, PHP/MySQL, Javascript, and Java. HTML5 and CSS are heavily used in web development, although these really aren't programming languages per se. You will also hear of newer languages, such as Objective C, Python, R, Ruby, Swift, Rust, Elixir and Scala. Specific languages tend to have their own areas of heaviest usage. C/C++/Visual C# are used for developing both applications software and operating systems; Visual Basic is used for business applications, SQL for database applications, PHP for web database applications, and Javascript for certain types of functionality in web development. Java (which has nothing to do with Javascript, by the way) is used both for the web on computers, as well as thousands of other computerized products such as parking meters and Internet-connected refrigerators. Other older languages still in use are FORTRAN (for scientific work), COBOL (for business work), PL/I (for business and scientific), SAS (for statistical) and GPSS (for simulations). C# (pronounced C-sharp) is a language developed by Microsoft, and is encased in the Visual Studio suite (which provides a complete development environment). It is based on C++, but is more powerful in many ways. Visual C# and Visual Basic are for Windows development only. C and C++ can be used for Windows and for Unix, Linux and the Mac OS. Hundreds more languages have existed and will continue to exist.

The computer cannot actually understand these languages directly. A program called a “compiler” translates the programs into a form the processor understands. We call that form “machine language”. Humans don’t often program directly in machine language (it’s all zeroes and ones, and way too difficult and tedious), but sometimes program in “assembly language” (which is very close to machine language, but easier to work with) to provide more precise control of what’s happening at the processor level. While assembly language programs run more efficiently, they are very inefficient to create and maintain. The languages we generally use (listed above) are called “high-level languages” and are the preferred way to develop.

Take a look at the video on Canvas to see how we go about taking a program from its high-level language (such as C) to the low-level language (machine or assembly) that results in the executable file that you actually run. The *editor* is what we use to enter our C program (called *source code*). Again, the *compiler* is the program that does the translation to a low-level language that we never see. The *linkage editor* does some final work that results in us having an executable (in Windows, this is a .EXE file). In today's world, we combine all of these programs into a single comprehensive *development environment* -- one program that does it all. In this part of the course, we use the Dev C++ Development Environment.

Why C and VB for this course?

In the 80s and 90s, we used PL/I ("P L one") and Pascal for this course. PL/I was an early attempt at a full-featured language, combining the best of COBOL and FORTRAN. Pascal was created in the early 1970s to provide a complete, yet more simplified and structured language for learning programming. It was felt that by learning the concepts of programming in this language, you could master them more easily and then move that knowledge to more advanced and difficult languages. Pascal had been used recently for application development for Apple Macintosh computers and small development projects for the DOS and Windows operating environments. It is primarily, though, a language used for teaching programming in high school, college, and technical academies.

Today, there is a hunger for tools to develop programs for the various Microsoft Windows environments (such as Windows 10, Windows Server, and Windows RT), the MacOS world, general web browsers, the tablet and smartphone world, and even more, including Internet-enabled devices like the Amazon Echo. Pascal is history for these new environments. Visual C# and Visual Basic stepped to the plate as the most straightforward tools for Windows. But of these languages have quirks and difficulties which make the absolute basics of elementary programming lost in the shuffle. And even Java, Javascript, PHP, Swift and Ruby step a little too quickly for a fundamentals course. We've opted to start your programming education with C. Python is also a worthy candidate, and we will probably utilize it here in the near future.

C is the most widely used procedural language (we'll discuss the differences between procedural and object-oriented in Module 7) and is great for learning the absolute fundamentals of a programming language. We will discuss only the most elementary areas of C so that you get a taste of straightforward traditional programming. You will need this knowledge to proceed to many different programming environments, including shell scripting, macro development, process control and basic business programming. If you examine almost every one of the other languages I've mentioned (especially the ones developed past 1990), they bear a close resemblance to C.

C# is a relatively new language created by Microsoft to supplant C++, an "object-oriented" version of C that has been popular since the 90s. While C++ is a language created to run in a variety of environments (such as UNIX, Linux and Windows), C# only runs in Microsoft Windows environments. It is fairly difficult to start your object-oriented programming education in such a rich and complex language, so we have opted to use an easier language--Visual Basic--to introduce programming in a Windows object-oriented event-driven environment.

Mac, Android and iOS users need not fear. The principles you learn here working with both C and Visual Basic carry over nicely to languages that are used in these environments.

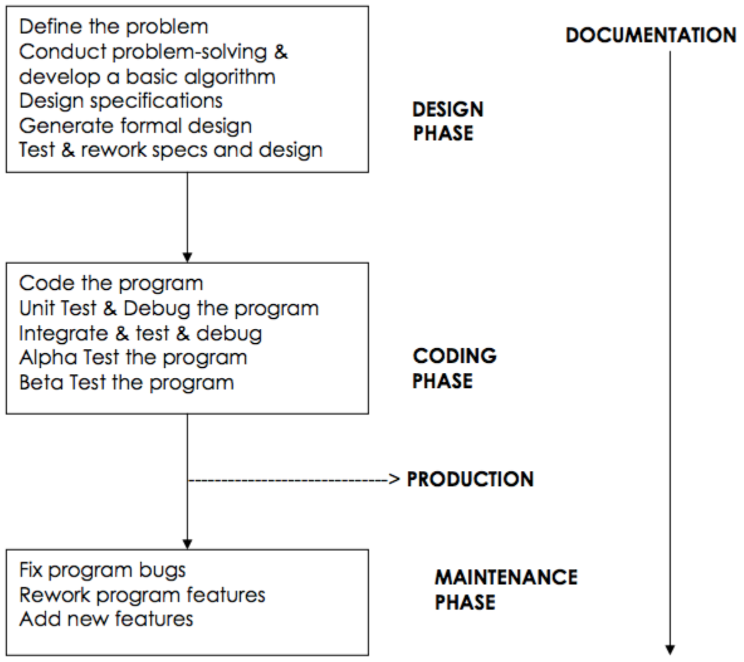
After learning the basic principles in C and Visual Basic (also called VB), you should find it much easier to master languages like C#, Java, Objective C and Swift.This course should not be seen as a C or VB programming language course per se; it is a course that introduces the fundamentals of programming. It is best to learn the basics before heading into more difficult languages.

The Software Development Process Overview

Software development has many distinct phases. It starts with a problem or need. The developers (programmers and systems analysts) then try to find a solution to the problem, and will shape that solution into a program. But like with other fields--such as construction--they do not simply sit down and start writing program code. They go through a structured process, in which they produce written specifications and then a written formal design (like a blueprint), prior to writing actual code. Then when after the code, a period of testing the code and getting the bugs out will occur. And when the software is “finished”, it will continue to be worked on due to changing user requirements, changing hardware, and changing operating system issues. This process is quite cyclical and is called the “Software Development Process”.

There are many different formal approaches, however, that we can label "Software Development Process". What I discuss below is a classic, traditional approach. There are more modern approaches that are popular in the development world today, and these are discussed in a separate lecture called Alternative Development Methodologies. While these are very popular and contemporary, I take a bit of a jaundiced view, having been in the I.T. world for all of my adult life. I've seen many methodologies come and go, almost similar to the various genres of popular music. You'll see Agile Development (an example of a popular contemporary approach) today, with its unique brand of terminology and approaches. I'd bet cash money, though, that 10 years from today some other methodology will supplant it. My advice? Learn how to learn the rapid and unyielding changing technologies in the I.T. and software development worlds.

We focus below on the approach that overlays most modern approaches in some way. Many people mistakenly refer to this as the "Waterfall" approach, but that's not entirely correct. Waterfall presupposes a progression through the various phases of development without the necessary cycling back and refinement of work done in each phase. Here we specifically mention that each phase and sub-phase below is cyclical and iterative, and that's not really Waterfall.



Notes on the above graphic: You don't see it on here, but imagine additional dotted lines flowing from each item in each box back to each of the previous items in all boxes. While the main flow goes down through the chart, the process is iterative. After completing each box, we MAY go ahead to the next, or due to a change in requirements (or after finding something non-optimal) we may have to "return to the drawing board" and head back to a previous step to refine it or start over.

Design Phase

In the design phase, the programmer (or possibly the systems analyst) initially works with the user to identify the problem. I always say that one shouldn't develop software in a vacuum. The software should do one or more of the following things: increase sales, decrease costs, or increase productivity. So the developer should work with the user to identify where there's a problem (in the above 3 categories) that needs to be solved. We'll talk in the next major section of this "lecture" about problem-solving.

After identifying the problem, the developer should work to develop a solution. If a systems analyst is involved, the solution might be a complete system of individual automated and manual procedures that work together and share data. If a programmer is involved, we are probably looking at a more limited solution that involves creation of a program or two or three. But this initial solution is not a program, per se, but a step-by-step process of what needs to be done. We call that process an algorithm. The term "algorithm" comes from the scientific world. If a systems analyst is involved, then he or she will work with the programmer to develop the algorithm. The programmer will be more concerned with the detailed part of the algorithm.

After developing the algorithm, one cannot assume that it works. The programmer must test the algorithm by following it step by step to see if it makes sense. We call this desk-checking the algorithm. It doesn't ensure that the program you develop later will be error-free, but it does show you if there are any fundamental flaws in the algorithm.

Depending on the complexity of the project, the developers might also work with the user to develop specifications. Specifications (or "specs") outline a variety of requirements that the solution will have to fulfill. They might include hardware and operating system requirements, user interface requirements, or functional items. Specs are generally developed after the problem has been thought about. They are usually outlined in a formal document, and often are not done until algorithm development is complete. We discuss specs in the last part of the lecture.

Following the development of the algorithm and the specs, the programmer creates a formal document called the program design or the design. The design is very close to a blueprint in construction. It may use a variety of diagrams to show the overall organization of the program and how it flows logically. A programmer will develop the actual program using the design as a guide. Within the design phase, there can be some cycling back to the initial problem-solving phase if the developed algorithm doesn't properly or efficiently solve the problem. Many projects spend ½ of the entire development period doing design. If one does thorough design work, then the chance of program development going smoothly is greatly increased. Back in college, my friend Sylvana used to spend days doing her design, while I and all the other guys sometimes did our macho thing and tried to bypass this crucial step. Almost always, once Sylvana got to the next phase--coding--she had great success in quickly getting her programs to run without errors. Us guys? Not a chance. Because we didn't spend time to design, we floundered, producing programs that didn't solve the problem and were full of bugs. We learned QUICKLY to not repeat that mistake in future classes. Do your design. It's like eating spinach.

The types of diagrams you will develop for design depends on the type of programming you'll be doing. For traditional programming and systems analysis, hierarchy charts, Nassi-Schneiderman charts, flowcharts, data-flow diagrams, and finite-state diagrams have been popular. In the new more sophisticated world of object-oriented, database-oriented and graphical programming, something called UML (Unified Modeling Language) was developed to set forth a new set of standards for a variety of design diagrams. Moreover, periodically someone develops a new design methodology with a fancy name and even fancier diagrams. You will take courses in these methodologies along with your programming language courses.

Coding Phase

In the coding phase, we actually develop the program in the programming language. You will find coding a bit difficult as you learn C and Visual Basic (or any other new language). But as you become acclimated to the language, coding gets easier and easier. The difficult part of development really is the original design. As you "code" the program, your programming editor will often help you by automatically highlighting your code contextually, pointing out some (but not all) errors in punctuation and spelling. After writing the code, you will compile the code. If you have compilation errors, you must fix these prior to actually executing the program. This is a cyclical process, and highly cyclical when you are first learning programming. Once the program compiles successfully, you then begin the long, arduous task of debugging the program. Sometimes you will have obvious execution errors. Other times, errors may not crop up until you do some testing. Whatever the case, you will generally spend a lot of time in this sub-phase. Do you know how many errors an early version of Microsoft Word (version 6.0 to be exact) had when it was first completed (prior to release)? 20,000 documented bugs! And during the public testing and debugging phase, this number was whittled down, but was still a robust 8,000! Imagine a product to be used by millions of people for everyday work having that many documented errors (and that number being "acceptable")! It illustrates the complexity of modern software development.

Not only will you test your programs, but you will also employ test engineers to test your programs too. You will test them from a programmer perspective, while test engineers will test this against the specifications. Later this quarter we'll discuss this long and thorough period of formalized testing. It's critically important and lack of such testing virtually guarantees that your program will be unstable and frustrating to its users due to errors and crashes. Sometimes your coding will unveil problems in the original design. Perhaps it was inefficient or resource-heavy. If this is the case, then it's back to redoing the design (hence the phrase, "Back to the drawing board!").

Production

When the program looks like it will run in a stable manner, and will satisfy the requirements of the users, it's ready to be released into production. The term "production" is used in a commercial sense (i.e., reproducing the disks and the manuals), but is also applied to in-house and consulting projects. If you give or sell the program to the user and say "Use It!", then the program has been placed in production. It should be noted that the program is not bug-free when in production. It is merely considered "acceptable for release".

Maintenance Phase

While the program is in production, programmers continue to work on it to improve efficiency, eliminate additional bugs, and add features. This period is called the maintenance phase. Sometimes the operating system or business conditions will change, forcing a slight “tweak” of the software. This reconfiguration of the software is another task the programmer performs in the maintenance phase. Commercial software vendors usually release one or two slightly improved versions of the software after the main release (within a single version, they also offer numerous "releases"). They do this for free or for a nominal charge. However, because they don’t make money unless the sell newer, bigger, better versions, they usually pack the software with additional features (many of which users often don’t care about) so they can release a version upgrade that costs a substantial amount of money. The amount is generally less than the original price (generally ¼ of the original), but still substantial enough to make a profit. The programmer should design the program in such a way that allows for easy reconfiguration. We call this good software engineering. However, even if someone engineers the program well, a visit back to the design phase will probably still be necessary if more than trivial changes in the program are made.

Documentation

Documentation is a very broad term, encompassing all written materials developed in conjunction with a program. The specs, formal design, code listings, programmer comments within the code listings, users manuals, user reference, and tutorial materials are all part of the documentation. The documentation is produced right alongside the three phases (plus production). User manuals and references materials can’t generally be developed until the specs are complete. The skills required to develop and maintain documentation vary based on the type of documentation being produced. User manuals and tutorials would require a training/educational background, with solid writing and editing skills. Some specs are best produced by an analyst with good business writing skills, while other portions are developed by the programmer. Sometimes a person on the programming team is responsible for keeping track of all the versions of documentation and code, in case the programmers need to go back to a version. That person is called the configuration librarian. Documentation is an important part of any development project.

Summary (Part 1)

This set of notes discussed different kinds of programming and programming languages. Some of the important points restated and simplified are:

- Programmers write programs based on specifications provided to them by systems analysts.
- Systems analysts work from business requirements and have a "user" level view of things.
- Programs must be tested to see how well they match the specifications; a category of people called Test Engineers perform this duty.

- Software can be roughly divided into two big categories: applications and systems. Applications do people things, while systems software makes sure that the computer and its peripherals operate.
- Programming languages are used to develop programs. There are many programming languages that have been created since the beginning of IT.
- C is like a "master language". Learn it and it will be much easier to learn most other modern languages. `The skills a programmer learns when mastering the fundamentals can be applied to virtually every language.
- The Software Development Process has many shapes and forms, but but virtually all deal with phases related to Design, Coding (writing the program), and Maintenance.
- Documentation is developed alongside program code and must meet the specifications also.

Part 2

Alternative Development Methodologies

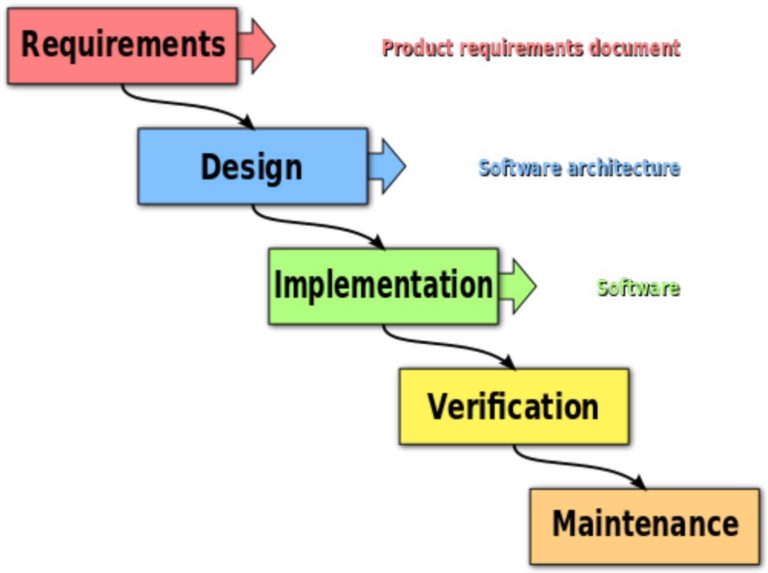
The Software Development Process described in the course notes and lecture is a generalized process that been in use for decades. Over this time, scholars in the IT field have developed a wide variety of development methodologies based loosely on this generalized process. These methodologies go in and out of vogue, and all have their plusses and minuses. As a student in this field, you will take courses and seminars which detail each the methodologies that are in effect in the industry at the time. Please realize that all of these have had merit and none should be fully discredited. There is always a "latest greatest" methodology. What is important is which methodology your company and/or projects are using. And when you become a Project Leader, you can determine what will work best for your projects.

This section briefly describes three contrasting methodologies in use today: Waterfall development, Rapid Application Development (RAD) and Agile Software Development (a specific subset of RAD). *Most of the text from this section is culled directly from various Wikipedia pages, which themselves have various credits and references. For easy readability, this document contains no citations and weaves together this information using my own statements. Please refer to the individual Wikipedia articles on the three methodologies for the specific citations and for more information on each methodology.*

Waterfall Development

The **Waterfall** model is a sequential software development process, in which progress is seen as flowing steadily downwards (like a waterfall) through the phases of Conception, Initiation, Analysis, Design (validation), Construction, Testing and maintenance.

On the right is a diagram of the unmodified "waterfall model". Progress flows from the top to the bottom, like a waterfall.



The Waterfall development model has its origins in the manufacturing and construction industries; highly structured physical environments in which after-the-fact changes are prohibitively costly, if not impossible. Since no formal software development methodologies existed at the time, this hardware-oriented model was simply adapted for software development.

The first formal description of the waterfall model is often cited to be an article published in 1970 by Winston W. Royce (1929–1995), although Royce did not use the term "waterfall" in this article. Royce was presenting this model as an example of a flawed, non-working model (Royce 1970). This is in fact the way the term has generally been used in writing about software development—as a way to criticize a commonly used software practice.

The waterfall model is argued by many to be a bad idea in practice. This is mainly because of their belief that it is impossible for any non-trivial project to get one phase of a software product's lifecycle perfected, before moving on to the next phases and learning from them.

For example, clients may not be aware of exactly what requirements they need before reviewing a working prototype and commenting on it; they may change their requirements constantly. Designers and programmers may have little control over this. If clients change their requirements after the design is finalized, the design must be modified to accommodate the new requirements. This effectively means invalidating a good deal of working hours, which means increased cost, especially if a large amount of the projects resources has already been invested in Big Design Up Front.

Designers may not be aware of future implementation difficulties when writing a design for an unimplemented software product. That is, it may become clear in the implementation phase that a particular area of program functionality is extraordinarily difficult to implement. If this is the case, it is better to revise the design than to persist in using a design that was made based on faulty predictions and that does not account for the newly discovered problem areas.

Even without such changing of the specification during implementation, there is the option either to start a new project from a scratch, "on a green field", or to continue some already existing, "a brown field" (from construction again). The Waterfall methodology can be used for continuous enhancement, even for existing SW, originally from another team. As well as in the case when the system analyst fails to capture the customer requirements correctly, the resulting impacts on the following phases (mainly the coding) still can be tamed by this methodology, in practice: A challenging job for a QA team.

While the diagram above slightly resembles the one drawn for the generalized Software Development Process we discussed earlier, note that this particular diagram shows no flexibility (as was described above). Because software development is highly fluid and iterative, many over the years have modified the Waterfall model to address these natural attributes. The “Sashimi Model”, for instance, is a Waterfall derivative. It is also known as "Waterfall model with overlapping phases" or "Waterfall model with feedback". The various Waterfall derivatives do attempt to address the rigidity of the original Waterfall model, and in many respects, can be considered effective on several levels.

Rapid Application Development (RAD)

Rapid Application Development (RAD) refers to a type of software development methodology that uses minimal planning in favor of rapid prototyping. The "planning" of software developed using RAD is interleaved with writing the software itself. The lack of extensive pre-planning generally allows software to be written much faster, and makes it easier to change requirements. This development methodology was introduced in the 1990s.

Rapid Application Development involves techniques like iterative development and software prototyping. According to Jeffrey Whitten in his 2004 text on systems analysis and design methods, it is a merger of various structured techniques, especially data-driven Information Engineering, with prototyping techniques to accelerate software systems development.

In Rapid Application Development, structured techniques and prototyping are especially used to define users' requirements and to design the final system. The development process starts with the development of preliminary data models and business process models using structured techniques. In the next stage, requirements are verified using prototyping, eventually to refine the data and process models. These stages are repeated iteratively; further development results in "a combined business requirements and technical design statement to be used for constructing new systems".

RAD approaches may entail compromises in functionality and performance in exchange for enabling faster development and facilitating application maintenance.

The shift from traditional session-based client/server development to open sessionless and collaborative development like Web 2.0 has increased the need for faster iterations through the phases of the Systems Development Life Cycle. This, coupled with the growing utilization of open source frameworks and products in core commercial development, has, for many developers, rekindled interest in finding a silver bullet RAD methodology.

Although most RAD methodologies foster software re-use, small team structure and distributed system development, most RAD practitioners recognize that, ultimately, there is no single “rapid” methodology that can provide an order of magnitude improvement over any other development methodology.

Various flavors of RAD have sprouted over the past 2 decades. Some of these are Agile Software Development, Extreme Programming (XP), Joint Application Development (JAD), Lean Software Development (LD), and Scrum.

All flavors of RAD have the potential for providing a good framework for faster product development with improved code quality, but successful implementation and benefits often hinge on project type, schedule, software release cycle and corporate culture. It may also be of interest that some of the largest software vendors such as Microsoft and IBM do not extensively utilize RAD in the development of their flagship products and for the most part, they still primarily rely on traditional Waterfall methodologies with some degree of spiraling.

Since rapid application development is an iterative and incremental process, it can lead to a succession of prototypes that never culminate in a satisfactory production application. Such failures may be avoided if the application development tools are robust, flexible, and put to proper use. This is addressed in methods such as the 2080 Development method or other post-agile variants.

Agile Software Development

Agile software development refers to a group of software development methodologies based on iterative development, where requirements and solutions evolve through collaboration between self-organizing cross-functional teams. The term was coined in the year 2001 when the Agile Manifesto was formulated.

Agile methods generally promote a disciplined project management process that encourages frequent inspection and adaptation, a leadership philosophy that encourages teamwork, self-organization and accountability, a set of engineering best practices intended to allow for rapid delivery of high-quality software, and a business approach that aligns development with customer needs and company goals.

Conceptual foundations of this framework are found in modern approaches to operations management and analysis, such as lean manufacturing, soft systems methodology, speech act theory (network of conversations approach), and Six Sigma.

There are many specific agile development methods. Most promote development iterations, teamwork, collaboration, and process adaptability throughout the life-cycle of the project. The most popular agile method today is called **Scrum**.

Agile methods break tasks into small increments with minimal planning, and do not directly involve long-term planning. Iterations are short time frames ("timeboxes") that typically last from one to four weeks. Each iteration involves a team working through a full software development cycle including planning, requirements analysis, design, coding, unit testing, and acceptance testing when a working product is demonstrated to stakeholders. This helps minimize overall risk, and lets the project adapt to changes quickly. Stakeholders produce documentation as required. An iteration may not add enough functionality to warrant a market release, but the goal is to have an available release (with minimal bugs) at the end of each iteration Multiple iterations may be required to release a product or new features.

Team composition in an agile project is usually cross-functional and self-organizing without consideration for any existing corporate hierarchy or the corporate roles of team members. Team members normally take responsibility for tasks that deliver the functionality an iteration requires. They decide individually how to meet an iteration's requirements.

Agile methods emphasize face-to-face communication over written documents when the team is all in the same location. When a team works in different locations, they maintain daily contact through videoconferencing, voice, e-mail, etc.

Most agile teams work in a single open office (called bullpen), which facilitates such communication. Team size is typically small (5-9 people) to help make team communication and team collaboration easier. Larger development efforts may be delivered by multiple teams working toward a common goal or different parts of an effort. This may also require a coordination of priorities across teams.

No matter what development disciplines are required, each agile team will contain a customer representative. This person is appointed by stakeholders to act on their behalf and makes a personal commitment to being available for developers to answer mid-iteration problem-domain questions. At the end of each iteration, stakeholders and the customer representative review progress and re-evaluate priorities with a view to optimizing the return on investment and ensuring alignment with customer needs and company goals.

Most agile implementations use a routine and formal daily face-to-face communication among team members. This specifically includes the customer representative and any interested stakeholders as observers. In a brief session, team members report to each other what they did yesterday, what they intend to do today, and what their roadblocks are. This standing face-to-face communication prevents problems being hidden.

Agile emphasizes working software as the primary measure of progress. This, combined with the preference for face-to-face communication, produces less written documentation than other methods—though, in an agile project, documentation and other artifacts rank equally with a working product. The agile method encourages stakeholders to prioritize wants with other iteration outcomes based exclusively on business value perceived at the beginning of the iteration.

Specific tools and techniques such as continuous integration, automated or Unit test, pair programming, test driven development, design patterns, domain-driven design, code refactoring and other techniques are often used to improve quality and enhance project agility.

Scrum

The most popular Agile approach today is Scrum. As with many new methodologies, many new terms are used for what are traditional development concepts. Is it effective? Only time will tell, but many similar techniques have risen and fallen over the past few decades (along with changes in fashion and what we use for listening to music). However, the concepts are sound, and as a developer you should be aware of this approach and its terminology since it will no doubt be used in your hiring interviews!

Please enjoy this YouTube video overview of Scrum.

Summary (Part 2)

I have worked with teams and projects that have employed a variety of the methodologies discussed above. Which is best? Well, that's like getting a bunch of consumers together to discuss which car is "best". All these methodologies have been employed in a great many companies and quite successfully. In general, newer generations of methodologies are more cost-effective and nimble, but it is often difficult to measure true effectiveness since a project team only employs a single methodology on a particular project. Since I've been in the I.T. field, I've seen countless approaches come and go, with fervent supporters of each variety. No doubt, a combination of corporate profitability and "the grass is greener on the other side" will drive us to newer and newer methods that come along. My suggestion is to learn how to work in a variety of these methodologies so that you know the lingo and the processes when you interview for project and career positions. Your ongoing discussions with industry developers and recruiters will give you direction on this.