10.3: Software Development

Many of the methodologies discussed above are used to manage software development since programming is complex, and sometimes errors are hard to detect. We learned in chapter 2 that software is created via programming, and programming is the process of creating a set of logical instructions for a digital device to follow using a programming language. The programming process is sometimes called “coding” because the syntax of a programming language is not in a form that everyone can understand – it is in “code.”

The process of developing good software is usually not as simple as sitting down and writing some code. True, sometimes a programmer can quickly write a short program to solve a need. But most of the time, the creation of software is a resource-intensive process that involves several different groups of people in an organization. In the following sections, we are going to review several different methodologies for software development.

Sidebar: The project management quality triangle

When developing software or any product or service, there is tension between the developers and the different stakeholder groups, such as management, users, and investors. When developing software, project managers must make tradeoffs between scope, schedule, and cost. This is illustrated by the project management quality triangle (Figure 10.3.1). For example, adding new features without adjusting timeline or resources will reduce quality. Understanding these constraints can help managers make wise tradeoffs when planning software projects.

Figure 10.3.1 illustrates the tension of the three requirements: time, cost, and quality that project managers need to make tradeoffs in. From how quickly the software can be developed (time), to how much money will be spent (cost), to how well it will be built (quality). The quality triangle is a simple concept. It states that you can only address two of the following: time, cost, and quality for any product or service being developed.

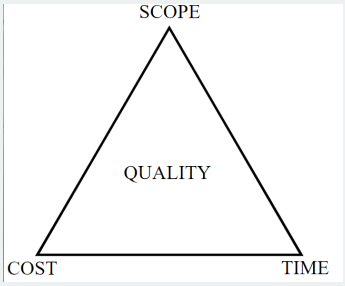

Figure 10.3.1: Project Management Quality Triangle. Image by Mapto is licensed Public domain

So what does it mean that you can only address two of the three? It means that the finished product's quality depends on the three variables: scope, schedule, and the allocated budget. Changes in any of these three variables affect the other two, hence, the quality.

For example, if a feature is added, but no additional time is added to the schedule to develop and test, the code's quality may suffer, even if more money is added. There are times when it is not even feasible to make the tradeoff. For example, adding more people to a project where members are so overwhelmed that they don’t have time to manage or train new people. Overall, this model helps us understand the tradeoffs we must make when developing new products and services.

10.3.1: Software Development Teams

Software development involves coordinators and specialists playing different roles, discussed in the earlier chapter, on the team:

- Project managers - oversee the overall project timeline, budget, resource planning, and cross-functional communication.
- Software developers/engineers - develop code and programs based on the specifications.
- Quality assurance testers - test the software to identify defects and ensure requirements are met.
- User experience (UX) designers - design intuitive and user-friendly interfaces.
- Business analysts - document business requirements that software should fulfill.
- End user representatives - provide ongoing feedback during development to ensure software meets needs.

Designing an optimal team is essential for a successful implementation.

10.3.2: Programming Languages

One of the important decisions that a project team needs to make is to decide which programming language(s) are to be used and associated tools in the development process. As mentioned in chapter 3, software developers create software using one of several programming languages. A programming language is a formal language that provides a way for a programmer to create structured code to communicate logic in a format that the computer hardware can execute. Over the past few decades, many different programming languages have evolved to meet many different needs.

There is no one way to categorize the languages. Still, they are often grouped by type (i.e., query, scripting), or chronologically by year when it was introduced (i.,e. Fortran was introduced in 1954s), by their "generation," by how it was translated to the machine code, or how it was executed. We will discuss a few categories in this chapter.

10.3.2.1: Generations of Programming Languages

Early languages were specific to the type of hardware that had to be programmed; each type of computer hardware had a different low-level programming language (in fact, even today, there are differences at the lower level, though higher-level programming languages now obscure them). In these early languages, precise instructions had to be entered line by line – a tedious process.

Some common characteristics are summarized below to illustrate some differences among these generations:

	First-generation (1GL)	Second-generation (2GL)	Third-generation (3GL)	Fourth-generation (4GL)	Fifth-generation (5GL)
Time introduced (est).	1940s or earlier	1950s	1950s-1970s	1970s-1990s	1980s-1900s
Instructions	They are made of binary numbers of 0s and 1s	Use a set of syntax that is readable by human and programmers	The syntax is more structured and is made up of more human-like language	The syntax is friendly to non-programmers	Still in progress.
Category	Machine dependent Machine code	Machine dependent Low level, Assembly Languages	Machine independent High Level	Machine independent High-level abstraction, Advanced 3GLs	Logic programming
Advantage	Very fast, no need for 'translation' to 0s and 1s	Code can be read and written by programmers easier than learning machine code	More machine-independent More friendly to programmers General-purpose	Easy to learn	May not need programmers to write programs
Disadvantage	Machine dependent, not portable	Must be converted to machine code, still machine-dependent	May go multiple steps to translate to machine code	More specialized	Still early in the adoption phase
Today's usage	If needed to interact with hardware directly such as drivers (i.e., USB driver)	If needed to interact with hardware directly such as drivers (i.e., USB driver)	Modern 3GLs are more commonly used. Early 3GLs are used to maintain existing business programs or scientific programs	Database, web development	Limited Visual tools, Artificial intelligence research

	First-generation (1GL)	Second-generation (2GL)	Third-generation (3GL)	Fourth-generation (4GL)	Fifth-generation (5GL)
Examples	Machine language	Assembly language	Early 3GLs: COBOL, Fortran Modern 3PLs: C, C++, Java, Javascript	Perl, PhP, Python, SQL, Ruby	Mercury, OPS5

Statista.com reported that by early 2020, Javascript was the most used language among developers worldwide. To see the complete list, please visit Statista.com for more details.

Examples of languages

First-generation language: machine code. In machine code, programming is done by directly setting actual ones and zeroes (the bits) using binary code. Here is an example program that adds 1234 and 4321 using machine language:

10111001	00000000
11010010	10100001
00000100	00000000
10001001	00000000
00001110	10001011
00000000	00011110
00000000	00011110
00000000	00000010
10111001	00000000
11100001	00000011
00010000	11000011
10001001	10100011
00001110	00000100
00000010	00000000

Second-generation language. Assembly language gives English-like phrases to the machine-code instructions, making it easier to program. An assembly-language program must be run through an assembler, which converts it into machine code. Here is an example program that adds 1234 and 4321 using assembly language:

MOV CX,1234 MOV DS:[0],CX MOV CX,4321 MOV AX,DS:[0]

MOV BX,DS:[2] ADD AX,BX

MOV DS:[4],AX

Third-generation languages are not specific to the type of hardware they run and are much more like spoken languages. Most third-generation languages must be compiled, a process that converts them into machine code. Well-known third-generation languages include BASIC, C, Pascal, and Java. Here is an example using BASIC:

A=1234 B=4321 C=A+B END

Fourth-generation languages are a class of programming tools that enable fast application development using intuitive interfaces and environments. Many times, a fourth-generation language has a particular purpose, such as database interaction or report-writing. These tools can be used by those with very little formal training in programming and allow for the quick development of applications and/or functionality. Examples of fourth-generation languages include Clipper, FOCUS, FoxPro, SQL, and SPSS.

Why would anyone want to program in a lower-level language when they require so much more work? The answer is similar to why some prefer to drive stick-shift automobiles instead of automatic transmission: control and efficiency. Lower-level languages, such as assembly language, are much more efficient and execute much more quickly. You have finer control over the hardware as well. Sometimes, a combination of higher- and lower-level languages is mixed together to get the best of both worlds: the programmer will create the overall structure and interface using a higher-level language but will use lower-level languages wherever in the program that requires more precision.

10.3.2.2: Compiled vs. Interpreted

Besides classifying a programming language based on its generation, it can also be classified as compiled or interpreted language. As we have learned, a computer language is written in a human-readable form. In a compiled language, the program code is translated into a machine-readable form called an executable that can be run on the hardware. Some well-known compiled languages include C, C++, and COBOL.

An interpreted language requires a runtime program to be installed to execute. This runtime program then interprets the program code line by line and runs it. Interpreted languages are generally easier to work with but are slower and require more system resources. Examples of popular interpreted languages include BASIC, PHP, PERL, and Python. The web languages such as HTML and Javascript would also be considered interpreted because they require a browser to run.

The Java programming language is an interesting exception to this classification, as it is actually a hybrid of the two. A program written in Java is partially compiled to create a program that can be understood by the Java Virtual Machine (JVM). Each type of operating system has its own JVM, which must be installed, allowing Java programs to run on many different types of operating systems.

10.3.2.3: Procedural vs. Object-Oriented

A procedural programming language is designed to allow a programmer to define a specific starting point for the program and then execute sequentially. All early programming languages worked this way. As user interfaces became more interactive and graphical, it made sense for programming languages to evolve to allow the user to define the program's flow. The object-oriented programming language is set up to define “objects” that can take certain actions based on user input. In other words, a procedural program focuses on the sequence of activities to be performed; an object-oriented program focuses on the different items being manipulated.

For example, in a human-resources system, an “EMPLOYEE” object would be needed. If the program needed to retrieve or set data regarding an employee, it would first create an employee object in the program and then set or retrieve the values needed. Every object has properties, which are descriptive fields associated with the object. In the example below, an employee object has the properties “Name,” “Employee number,” “Birthdate,” and “Date of hire.” An object also has “methods,” which can take actions related to the object. In the example, there are two methods. The first is “ComputePay(),” which will return the current amount owed to the employee. The second is “ListEmployees(),” which will retrieve a list of employees who report to this employee.

Employee Object

Object: EMPLOYEE
First_Name Last_Name Employee_ID Birthdate Date_of_hire
ComputePay() ListEmployees()

10.3.3: Programming Tools

Another decision that needs to be made during the development of an IS is the set of tools needed to write programs. To write programs, programmers need tools to enter code, check for the code's syntax, and some method to translate their code into machine code. To be more efficient at programming, programmers use integrated tools such as an integrated development environment (IDE) or computer-aided software-engineering (CASE) tools.

10.3.3.1: Integrated Development Environment (IDE)

For most programming languages, an IDE can be used. An IDE provides various tools for the programmer, all in one place with a consistent user interface. IDE usually includes:

- an editor for writing the program that will color-code or highlight keywords from the programming language;
- a help system that gives detailed documentation regarding the programming language;
- a compiler/interpreter, which will allow the programmer to run the program;

- a debugging tool, which will provide the programmer details about the execution of the program to resolve problems in the code; and
- a check-in/check-out mechanism allows a team of programmers to work together on a project and not write over each other's code changes.

Statista.com reports that 80% of software developers worldwide from 2018 and 2019 use a source code collaboration tool such as GitHub, 77% use a standalone IDE such as Eclipse, 69% use Microsoft Visual Studio. For a complete list, please visit statista.com.

10.3.3.2: Computer-aided software engineering (CASE) Tools

While an IDE provides several tools to assist the programmer in writing the program, the code still must be written. Computer-aided software engineering (CASE) tools allow a designer to develop software with little or no programming. Instead, the CASE tool writes the code for the designer. CASE tools come in many varieties, but their goal is to generate quality code based on the designer's input.

10.3.3.3: Build vs. Buy or Subscribe

When an organization decides that a new software program needs to be developed, they must determine if it makes more sense to build it themselves or purchase it from an outside company. This is the "build vs. buy" decision. This 'buy' decision now includes the option to subscribe instead of buying it outright.

Listen to Ramli John as he shared 3 factors you should consider to help you decide which option is best for you. (warning: video contains Appcues product marketing)

There are many advantages to purchasing software from an outside company. First, it is generally less expensive to purchase a software package than to build it. Second, when a software package is purchased, it is available much more quickly than if the package is built in-house. Third, companies or consumers pay a one-time price and get to keep the software for as long as the license allows and could be as long as you own it or even after the vendor stops supporting it. Software applications can take months or years to build; a purchased package can be up and running within a month. A purchased package has already been tested, and many of the bugs have already been worked out, and additional support contracts can be purchased. It is the role of a systems integrator to make various purchased systems and the existing systems at the organization work together.

There are also disadvantages to purchasing software. First, the same software you are using can be used by your competitors. If a company is trying to differentiate itself based on a business process in that purchased software, it will have a hard time doing so if its competitors use the same software. Another disadvantage to purchasing software is the process of customization. If you purchase a software package from a vendor and then customize it, you will have to manage those customizations every time the vendor provides an upgrade. With the rise of security and privacy, companies may lack the in-house expertise to respond quickly. Installing various updates and dealing with bugs encountered may also be a burden to IT staff and users. This can become an administrative headache.

A hybrid solution is to subscribe. Subscribe means that instead of selling products individually, vendors now offer a subscription model that the users can rent and pay periodically, such as monthly, yearly. The renting model has been used in many other industries such as movies, books and recently has moved into high tech industries. Companies and consumers can now subscribe to almost everything, as we discussed in earlier chapters, from additional storage in your email platforms such as Google Drive or Microsoft Onedrive, to software such as Quickbooks, Microsoft Office 365, to hosting and web support services such as Amazon AWS. Vendors benefit from converting one-time sales to recurring sales and increase customer loyalty. Customers benefit from the headache of installing updates, having the software support and updates taken care of automatically, knowing that the software continues to be updated with new features. A subscription model is now a prevalent option for both consumers and businesses.

Even if an organization determines to buy or subscribe, it still makes sense to go through many of the same analyses to compare the costs and benefits of building it themselves. This is an important decision that could have a long-term strategic impact on the organization.

Use Case: Build vs. Buy 10.3.1

Company My Widgets is evaluating options for a new enterprise system to manage its supply chain and manufacturing operations. The COO has requested a build vs buy analysis.

Solution

1. Build Option: The IT department could develop a custom supply chain management system tailored to My Widgets' unique manufacturing workflows. They already have experience with the required technologies. This system would provide competitive advantages from proprietary processes. However, initial development is estimated at 18 months with ongoing maintenance.
2. Buy Option: Several vendors offer supply chain management software, such as SAP and Oracle. The systems provide standard functionality that covers 70% of My Widgets' needs. They could be implemented in 6 months but would require custom integrations. Vendor costs are $1M for licenses and support.

In this scenario, the company must weigh factors like competitive advantage, development costs, maintenance, and speed-to-implementation. While building provides customization, buying could allow faster rollout and leverage vendor expertise. Engaging users and mapping processes to packaged capabilities is essential in the decision-making process.

10.3.4: Web Services

Chapter 3 stated that the move to cloud computing has allowed software to be looked at as a service. One option companies have these days to license functions provided by other companies instead of writing the code themselves. These are called web services, and they can greatly simplify the addition of functionality to a website.

For example, suppose a company wishes to provide a map showing the location of someone who has called their support line. By utilizing Google Maps API web services, they can build a Google Map right into their application. Or a shoe company could make it easier for its retailers to sell shoes online by providing a shoe-size web service that the retailers could embed right into their website.

Web services can blur the lines between “build vs. buy.” Companies can choose to build a software application themselves but then purchase functionality from vendors to supplement their system.

10.3.5: End-User Computing or Shadow IT

In many organizations, application development is not limited to the programmers and analysts in the information-technology department. Especially in larger organizations, other departments develop their own department-specific applications. The people who build these are not necessarily trained in programming or application development, but they tend to be adept with computers. A person, for example, who is skilled in a particular software package, such as a spreadsheet or database package, may be called upon to build smaller applications for use by his or her own department. This phenomenon is referred to as ***end-user development***, or ***end-user computing***, or ***Shadow IT***.

Shadow IT is the use of IT-related hardware or software by a department or individual without the knowledge of the IT or security group within the organization. It can encompass cloud services, software, and hardware.

Some examples of shadow IT applications and cloud based services include:

- Productivity tools such as Asana and Trello
- Cloud storage, file-sharing and document-editing applications such as Google Drive, Dropbox, Google Docs
- Communication and messaging apps such as Skype, Slack, Whatsapp, Telegram, as well as personal email accounts.

End-user computing can have many advantages for an organization. First, it brings the development of applications closer to those who will use them. Because IT departments are sometimes quite backlogged, it also provides a means to have software created more quickly. Many organizations encourage end-user computing to reduce the strain on the IT department. for example, Excel can be used to create a customer-facing application that depicts dynamic views into different investment scenarios. An end-user development like this can help customers easily view, comprehend and determine their preferred course of action.

End-user computing does have its disadvantages as well. If departments within an organization are developing their own applications, the organization may end up with several applications that perform similar functions, which is inefficient since it duplicated effort. Sometimes, these different versions of the same application provide different results, bringing confusion when departments interact. These applications are often developed by someone with little or no formal training in programming. In these cases, the software developed can have problems that have to be resolved by the IT department. Shadow IT can also pose significant security risks. Because the IT team is unaware of shadow IT, it doesn't monitor these assets, or address their vulnerabilities. End-user computing can be beneficial to an organization, but it should be managed. The IT department should set guidelines and provide tools for the departments who want to create their own solutions.

Communication between departments will go a long way towards the successful use of end-user computing.

10.3.6: Mobile Platforms and Tools

There are a wide variety of development tools available for building mobile applications on different platforms:

- iOS apps are built using Swift or Objective-C languages and Xcode IDE, along with Apple's software development kit (SDK).
- Android apps are built using Java or Kotlin, along with Android Studio IDE and Android SDK.
- Cross-platform apps use frameworks like Xamarin, React Native, Flutter to allow development in languages like C# and JavaScript.
- Web apps use standard web development tools like HTML, CSS, JavaScript.

Understanding the different tools for different mobile platforms helps inform the build vs buy decision for organizations.

Sidebar: Building a Mobile App

Software development typically includes building applications to run on desktops, servers, or mainframes. However, the web's commercialization has created additional software development categories such as web design, content development, web server. Web-related development effort for the internet is now called web development. Earlier web development activities include building websites to support businesses or to build e-commerce systems and have made technologies such as HTML very popular with web designers and programming languages such as Perl, Python, Java popular for programmers. Pre-packaged websites are now available for consumers to purchase without learning HTML or hiring a web designer. For example, entrepreneurs who want to start a bakery business can now buy a pre-build website with a shopping cart, all ready to start a business without incurring costly expenses to build it themselves.

With the rise of mobile phones, a new type of software development called mobile app development came into being. Statista.com forecasts that Mobile apps revenues will increase significantly from $98B in 2014 to over $935B by 2023. This means that the need for mobile app developers has also increased.

In many ways, building an application for a mobile device is the same as building an application for a traditional computer. Understanding the application requirements, designing the interface, working with users – all of these steps still need to be carried out. The decision process to pick the right programming languages and tools remains the same.

However, there are specific differences that programmers must consider in building apps for mobile devices. They are:

- The user interface must vary to adapt to different screen size
- The use of fingers as pointers or to type in text instead of keyboard and mouse on the desktop
- Specific requirements from the OS vendor must be met for the app to be included in each store (i.e., Apple's App Store or Android's Play Store)
- The integration with the desktop or the cloud to synch up data
- Tight integration with other built-in hardware such as cameras, biometric or motion sensors.
- Less available memory, storage space, and processing power

Mobile apps are now available for just about everything and continue to grow.

10.3.7: Risks of End-User Computing

While end-user computing can provide organizations with more flexibility, there are also risks to consider:

- Unsupported apps - Apps developed by end users may not follow official development and documentation processes. This makes them harder to maintain when issues arise.
- Integration problems - End-user apps may not properly integrate with existing systems and databases. This can lead to data inconsistencies.
- Data discrepancies - With multiple departments creating their own apps, data can end up fragmented across siloed systems. The canonical source of truth is lost.
- Security vulnerabilities - Apps created outside official IT processes may not undergo proper security testing, leaving them open to cyberthreats.

To minimize risks from end-user computing, organizations should have IT oversight including:

- Published standards - Provide guidelines for end users on preferred programming languages, databases, etc.
- Code reviews - Require code reviews of end user apps to catch issues early.
- Change management - Formalize a process for requesting new end user apps and changes.
- Security testing - Test end user code for vulnerabilities before deployment.

With proper IT change management and governance, organizations can take advantage of end user computing benefits while reducing associated risks.

10.3.8: Security and Privacy in Software Development

While security applies to the entire software lifecycle, many key considerations need to be addressed up front during development. When developing new software applications, security and privacy need to be considered from the initial design phases:

- Secure design - Architect the application with security principles in mind from the start using best practices like encryption, least privilege access, and input validation.
- Data protection - Implement controls like encryption and tokenization to protect sensitive data in transit and at rest. Follow industry data security standards.
- Access controls - Limit user access with role-based permissions and multi-factor authentication. Maintain detailed logs for auditing.
- Patching - Ensure a patching roadmap is in place to quickly roll out new patches for security vulnerabilities as they arise.
- Testing - Perform extensive security testing activities like risk assessments, penetration testing, and code reviews to identify vulnerabilities.
- Compliance - Consider regulatory and industry-specific requirements like HIPAA and PCI DSS early in the process.

Making security and privacy a priority during software design, development, testing, and deployment reduces risk and instills trust.

10.3.9: Software Testing Methodologies

While testing does come before implementation, it is tightly aligned to development methodologies and software creation and the testing strategies should be part of the development methodologies and software creation. Some key testing methods include:

- Unit testing - Tests individual units of code like functions to make sure they work as intended. Confirms the smallest components operate correctly.
- Integration testing - Verifies that different modules and interfaces connect and interact properly. Makes sure components work together.
- System testing - Validates the entire system meets requirements. Tests the fully integrated system.
- User acceptance testing (UAT) - Real users test the software to validate it meets business needs and provides the expected user experience.
- Load testing - Checks application performance under expected user loads and identifies capacity limits.
- Security testing - Validates controls against vulnerabilities like exploits, data leaks, unauthorized access.

Managers should understand the purpose of each testing methodology to insure that the project meets the expected quality from the customers or users and are taken account in the budget planning.

10.3.10: Global Software Development

Many software projects today involve global teams with members distributed across multiple timezones and geographic regions. Effective collaboration is crucial for success and needs to be part of the planning of the development project. Considerations include:

- Communication - With remote team members, invest in tools for communication like Slack, Zoom, and email to bridge distance gaps.
- Collaboration - Use project management platforms like JIRA or Trello to coordinate tasks and status across locations. Version control with Git enables code sharing.
- Scheduling - Overlap team members working hours so collaboration can happen live. Schedule calls respecting time zones.
- Culture - Recognize cultural differences in communication styles and needs. Bridge language barriers.
- Onboarding - Train remote members on processes and tools. Create documentation assets for consistency.
- Bonding - Foster connections between remote members with team gatherings or visits if possible. Build relationships and trust.

With an intentional strategy and process for how to collaborate gobablly, the project will decrease the risk of failure during the implementation phase.

10.3.10.1: References:

Javascript was the most used language among developers worldwide (2020). Retrieved December 10, 2020, from Satistica.com

Google Maps Platform Documentation. Retrieved December 10, 2020, from https://developers.google.com/maps/documentation

Programming/development tools used by software developers worldwide from 2018 and 2019 (2020). Retrieved December 10, 2020, from Statista.com

Worldwide mobile app revenues in 2014 to 2023 (2010.) Retrieved December 10, 2020, from Statista.com

Shadow IT. IBM Topics. https://www.ibm.com/topics
