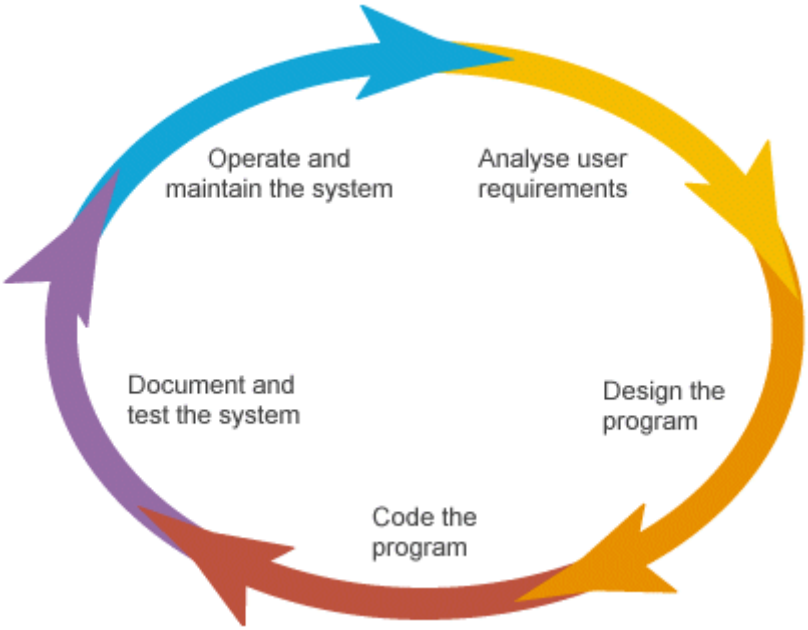
Topic: 4.4.1 Stages of software development

The Software Development Life Cycle is a process that ensures good software is built. Each phase in the life cycle has its own process and deliverables that feed into the next phase. There are typically 5 to 6 phases starting with the analysis and requirements gathering and ending with the implementation. Let's look in greater detail at each phase:



Requirements Gathering/Analysis

This phase is critical to the success of the project. Expectations (whether of a client or your team) need to be fleshed out in great detail and documented. This is an iterative process with much communication taking place between stakeholders, end users and the project team. The following techniques can be used to gather requirements:

Identify and capture stakeholder requirements using customer interviews and surveys.

Build multiple use cases to describe each action that a user will take in the new system.

Prototypes can be built to show the client what the end product will look like. Tools like Omnigraffle, HotGloo and Balsalmiq are great for this part of the process.

In a corporate setting, this means taking a look at your customers, figuring out what they want, and then designing what a successful outcome would look like in a new bit of software.

Topic: 4.4.1 Stages of software development

Design

Technical design requirements are prepared in this phase by lead development staff that can include architects and lead developers. The Business Requirements are used to define how the application will be written. Technical requirements will detail database tables to be added, new transactions to be defined, security processes and hardware and system requirements.

Let's look in more detail at some of the activities involved in this stage:

Risk analysis

Threats and vulnerabilities which may arise from interactions with other systems.

External or legacy code needs to be analyzed to determine if there are security vulnerabilities.

High-risk privacy projects could require review with a legal department. This review should consider what personal data to collect, how to collect it, and permissions/authorizations to make changes. This type of review is especially necessary with corporate projects.

Functional Specifications

Includes a description of interface requirements such as definition of data entry fields (allow numeric or alpha only, can it be left blank?)

Important details, like: can date entered be before current date? What timezone will user logins default to?

Workflow – after clicking approve button, which screen appears next?

Audit trail for every update on the database. This is where error monitoring and logging tools can be useful.

Non-Functional Specifications

Extensibility of the system – will current system easily allow new enhancements or features with the next rollout? This is critical for any application that you'll be adding new features and updating often.

Has the current or future capacity been analyzed for database requirements? Will the current build plan result in capacity issues shortly after you finish building?

Performance and response time – Has the expected response time been determined?

Resource Constraints – Are there constraints that need to be taken into consideration in this phase? Common ones include disk space, bandwidth, etc.

Topic: 4.4.1 Stages of software development

Coding

This phase is the actual coding and unit testing of the process by the development team. After each stage, the developer may demonstrate the work accomplished to the Business Analysts and tweaks and enhancements may be required. It's important in this phase for developers to be open-minded and flexible if any changes are introduced. This is normally the longest phase of the SDLC. The finished product here is input to the Testing phase.

Testing

Once the application is migrated to a test environment, different types of testing will be performed including integration and system testing. User acceptance testing is the last part of testing and is performed by the end users to ensure the system meets their expectations. At this point, defects may be found and more work may be required in the analysis, design or coding. Once sign-off is obtained by all relevant parties, implementation and deployment can begin.

Implementation/Deployment

The size of the project will determine the complexity of the deployment. Training may be required for end users, operations and on-call IT staff. Roll-out of the system may be performed in stages starting with one branch then slowly adding all locations or it could be a full blown implementation.

One of two methods can be followed in a SDLC process. Waterfall is the more traditional model and has a well-structured plan and requirements to be followed. This method works well for large projects that may take many months to develop. The Agile Methodology is more flexible in the requirements, design and coding process and is very iterative. This process works best for smaller projects and expectations of continuous improvement to the application. Whether you use one over the other will also depend to a large extent on the corporation and skills of the IT dept.

Documentation:

Documentation at every stage is necessary. That includes user and system documentation.

User documentation, this usually consists of:

- how to load/run the software
- how to save files
- how to do a search
- how to sort data
- how to do print outs
- how to add, delete or amend records
- the purpose of the system/program/software package
- screen layouts (input)
- print layouts (output)

03-111-222-ZAK | OlevelComputer AlevelComputer | @zakonweb | zak@zakonweb.com | www.zakonweb.com

Topic: 4.4.1 Stages of software development

- hardware requirements
- software requirements
- sample runs (with results and actual test data used)
- error handling/meaning of errors
- troubleshooting guide/help lines/FAQs
- how to log in/log out

Technical documentation, this usually consists of:

- program listing/coding
- programming language(s) used
- flowchart/algorithm
- purpose of the system/program/software
- input formats
- hardware requirements
- software requirements
- minimum memory requirements
- known “bugs” in the system
- list of variables used (and their meaning/description)
- file structures
- sample runs (with results and actual test data used)
- output formats
- validation rules

Phases may overlap:

Phases can and do overlap as previous phases are revisited, when more information becomes available. Phases overlap may also imply that there is no fixed finish between the start of one phase and the start of the next.

Topic: 4.4.1 Stages of software development

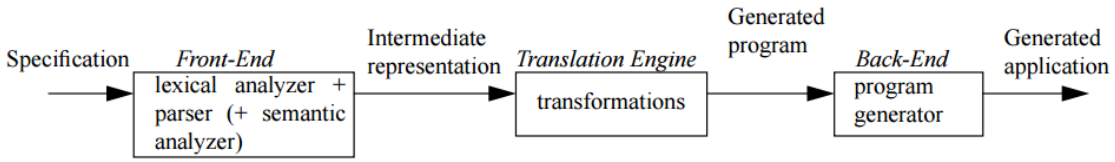
Program generator

Software program that enables an individual to create a program of their own with less effort and programming knowledge. With a program generator a user may only be required to specify the steps or rules required for his or her program and not need to write any code or very little code.

Some great examples of a program generator are Adventure Maker, Alice, Stagecast Creator, and YoYo Games.

When a programming activity is well-understood, it can be automated. Automation transforms software development from activities like rote coding and tedious debugging to that of specification, where the "what" of an application is declared and the "how" is left to a complex, but automatable mapping. Programs that perform such mappings are application generators (or just generators). In the technical sense, application generators are compilers for domain-specific programming languages (DSLs). There is no strict criterion for characterizing a language as "domain-specific" but the term is commonly used to describe programming languages for specialized tasks (as opposed to "general-purpose" programming languages).

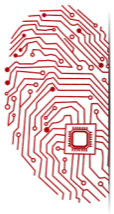
Examples are languages for implementing communication protocols, partial differential equation solvers, windowing software, etc. Although all compilers can be viewed as generators, generator research and practice has focused on problems different than those usually found in a classical treatment of compilers (e.g., Fig. below), such as programming language extensibility and program transformations.



A generator is similar to a conventional compiler, with a front-end, translation engine, and back-end.

Before we delve further into generator specifics, it is worth addressing the following question: why are generators needed?

Is it not sufficient to employ other programming tools (e.g., traditional software libraries)? One answer is that it is very hard to scale traditional tools to handle code that is highly complex, yet can be decomposed into simpler pieces in systematic ways. In this case, generators can be viewed as compact representations of software libraries of gigantic size—each library encoding all the useful code configurations that a generator can produce. With a generator, the same code is produced in a systematic way that allows a mechanical process to generate desired code configurations. Thus, for practical reasons, generators are the preferred way to represent large families of related applications. Another reason for using generators is that the specification languages that generators implement are much more concise and convenient than the language of the produced program (called the target language). The translation of specifications to target code is done correctly and quickly, thereby



Topic: 4.4.1 Stages of software development

substantially increasing programmer productivity. Further, generators can apply domain-specific optimizations (which are tedious and error-prone if done by hand) and can perform advanced error-checking (thereby automating correctness checks performed by domain experts). Compared to traditional software libraries, generators offer a much greater potential for optimization and can provide better error-checking.

Generators are gaining momentum in the software engineering community. In the past few decades, software construction has not seen any radical improvements with respect to increased productivity and reliability. The proponents of the generator approach consider generators to offer the greatest promise among emerging software technologies for the future of software development. In particular, advocates of generators consider them to be the right tool every time a software product is designed to be reused, or every time a domain exhibits significant systematic variability. This view promotes generators as a substitute for most, if not all, of the existing software libraries for appropriate domains.

In computing, a **visual programming** language (VPL) is any **programming** language that lets users create programs by manipulating **program** elements graphically rather than by specifying them textually. It is another form of program generators. Like visual basic dot net.

Topic: 4.4.1 Stages of software development

Program Libraries:

In computer science, a library is a collection of non-volatile resources used by computer programs, often to develop software. These may include configuration data, documentation, help data, message templates, pre-written code and subroutines, classes, values or type specifications.

Library code is organized in such a way that it can be used by multiple programs that have no connection to each other, while code that is part of a program is organized to only be used within that one program. The distinguishing feature is that a library is organized for the purposes of being reused by independent programs or sub-programs, and the user only needs to know the interface, and not the internal details of the library.

The value of a library is the reuse of the behavior. When a program invokes a library, it gains the behavior implemented inside that library without having to implement that behavior itself. Libraries encourage the sharing of code in a modular fashion, and ease the distribution of the code.

Most compiled languages have a standard library although programmers can also create their own custom libraries. Most modern software systems provide libraries that implement the majority of system services. Such libraries have commoditized the services which a modern application requires. As such, most code used by modern applications is provided in these system libraries.

Linkers link the libraries during runtime as they coexist with the language code. Object oriented programming classes are also available in the form of object's classes and are used by making a call to them with ordinary effort.