Recruiting Fundamentals Training

Software Development Life Cycle
An Overview

IT 4 Recruiters

IT4Recruiters.com Confidential

Revised: Rob Broadhead

IT4Recruiters.com
Suite 300 #288
115 Penn Warren Dr.
Brentwood, TN 37027-5054

Contact Name: Rob Broadhead (rob@it4recruiters.com)

Printed in the United States

Overview

SDLC is the acronym for “software development life cycle” or “systems development life cycle”. SDLC describes how to take a software product or project from idea to implementation and beyond. Fundamentally, SDLC is an attempt to make software product development much like developing a real world, physical product. The SDLC provides a form of assembly line of steps to go from idea to deployment, just as an assembly line might be used to create any widget. Software development is considered by SDLC advocates to be something that should be a reproducible process. SDLC is an attempt to define that reproducible process in a general manner to show how similar software development projects are to each other.

SDLC stands for “Software Development Life Cycle”

It provides a methodology for building software

The discussion of SDLC involves a number of areas where the opinions of the speaker will bias the presentation and this document is no different. Typical pros and cons will be highlighted where useful, but in other cases it will be left as an exercise to the reader as to what sort of opinions exist about a given facet of SDLC. When in doubt about how a methodology views the various SDLC steps review the methodology documentation. There will often be model approaches and reference projects to show how the methodology should look when put into action.

This applies to all forms of software development from web applications to mission control systems.

There are a number of different approaches to the life cycle, but all of them include the following steps:

- Gather and analyze requirements
- Design
- Implementation (coding)
- Testing
- Deployment
- Maintenance

A “cycle” of SDLC usually directly relates to a release of a version of software and sometimes an SDLC cycle is run through for each fix or patch release as well.

The “cycle” part of SDLC comes in as a product is enhanced or new features are added. At that point, the cycle starts again at the requirements step. This document will discuss each of these steps in more detail later in the document.

Keywords

Here are some useful definitions to keep in mind while reading this document:

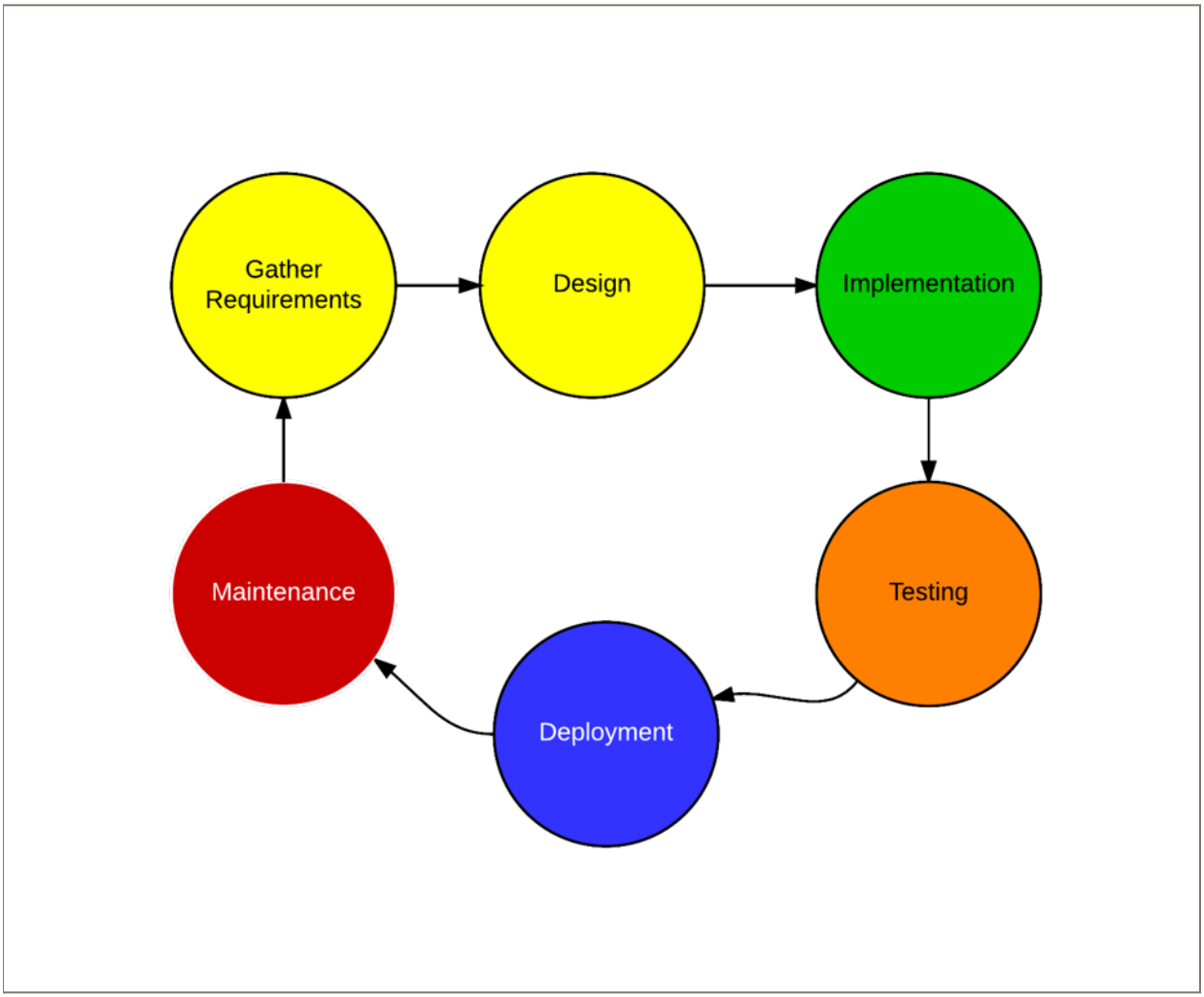
- Agile: An SDLC approach that is characterized by its avoidance of “administrative” tasks over “actual work”. The approach is summed up at http://agilemanifesto.org and has spawned a number of offshoots and varied approaches.
- Waterfall: A common and well documented SDLC approach that completes each step in its entirety before moving to the next one. Hence the term “waterfall.”
- Yada: “Yet another document artifact” this is just a convention used here.
- Methodology: Simply put, methodology is an approach to performing a task. In the SDLC context a methodology describes how you complete the steps defined by SDLC.
- Weasels: This term comes from Scott Adams the creator of the Dilbert comic strip and here refers to candidates that try to use certain language or experience to work their way into a position they are not qualified to fill.

You can find these and more at: http://it4recruiters.com/definitions.php.

SDLC Snapshot

SDLC has been around since the 1960s according to multiple sources, but it is not possible to pinpoint where SDLC as a whole started. It is an idea that grew out of some early "best practices" in software development and at some point the SDLC steps were put together as a process. It gained a near universal acceptance in the 1980s and became a facet of most development efforts. As software development approaches have grown and changed over the last 35 years, people have tinkered with aspects of the SDLC. The core steps exist in some form in most of the modern software development approaches. Each approach, however, has its own take on SDLC and the differences tend to be either in scope of the SDLC for each cycle or differences in emphasis on the various steps.

SDLC grew out of best practices for development and continues to advance and evolve today.



The image above shows the typical flow for a project. The process starts at "Gather Requirements" in most projects. The yellow circles are the steps that prepare for, and define, implementation. Implementation is often seen as the "meat" of the cycle as this is where the software is built and the coding tasks get done. The orange testing circle is a step that sometimes is barely addressed, and at other times is as big a part of the effort as the implementation step. Design and testing often overlap with other steps rather than being done in a silo. Deployment is often the end goal of the project and the "go live" date is tied to the completion of deployment. The cycle wraps up at the maintenance step as the product is either kept up to date in maintenance mode, or is enhanced and a new cycle is begun. Sometimes maintenance does not occur at all and the product goes right into the next iteration.

Gather Requirements:
Decide what to build and what it should do.

Design:
Decide how to build it

Implementation:
Build the system or application.

Testing:
Verify what was built matches the requirements

Deployment:
Deliver the product or system to the users

Maintenance:
Make adjustments or minor enhancements based on feedback from users and system monitoring.

There is a "religious" aspect to SDLC approaches just as there is to many other technology facets including languages (Java vs C), environments (Windows vs Mac vs UNIX), vendors (Google vs Apple), and anywhere an IT related choice is available. This can make learning about SDLC (or any other IT topic) difficult, and even confusing, as it is not always discussed in a purely rational light. The source sites and documents for the various approaches do tend to describe a sort of meta-SDLC though and that is where the SDLC steps in this document come from. There is not currently a standards body that defines how to approach SDLC in a general sense, it is left to each approach or implementor. This means a definition of SDLC for the Waterfall approach may seem foreign to a definition using the Agile approach. There are a number of links at the end of this document to help those that want to go deeper into this topic and learn more about the approaches in use.

There are a large number of user groups, websites, blogs, podcasts, etc that discuss SDLC and the various approaches. These also tend to have a large amount of traffic. A Google search of an approach and a key word such as "SDLC" and "tenets", "overview", "introduction", or "certification" will usually provide enough information to lead you to days of research and reading. IT4Recruiters also has focused papers on topics such as waterfall, Agile, and other approaches on the roadmap so check our website regularly for materials or projected publishing dates.

There are many sources for discussion and learning more about SDLC on the web

The key to understanding SDLC, no matter what approach is used, is to understand the six core SDLC steps. Lets look at each of those in detail...

Requirements gathering and documentation is arguably the most important phase, but also one that is often overlooked or at least given short shrift. At a high level, requirements gathering involves discussion with the customer or end user about what the end product should be. The customer may be a business stakeholder, a product manager, an advisory board, or any other person or group that can be considered the subject matter expert for the product. The most important deliverable of this phase is a communication (document, presentation, conference call, etc) from the "customer" to the implementation team. The communication provides details about what is to be produced so that the implementation team understands what they need to create.

The requirements step is often accomplished by documenting the features and functions that will define the product. The size and depth of this document varies widely from organization to organization and from approach to approach. The scope of the requirements also vary by methodology. There are methodologies that require all of the product requirements to be "set in stone" during this step. Some require only enough requirement definition to get the implementation started and the details will be filled in later. The remaining methodologies fall somewhere in between the two extremes. Good requirements are a key ingredient for good testing and quality analysis, as we will discuss later. A "correct" solution that has been implemented will fulfill all of the documented requirements.

IT projects often suffer from "doing anything is better than doing nothing" and this is why requirements gathering and definition is such a troublesome step. Experience has shown that it is better to "measure twice and cut once", so there is a value in gathering and defining requirements as well as "testing" them. This should include a sort of walk-through of the requirements that serves to test that they are reasonably complete and to avoid areas where requirements might paint one into a proverbial corner. This emphasis on requirements is not a majority view as requirements gathering can be costly and frustrating. Code is valued over planning and design in many of the companies as it is easier to measure. There is often little value seen in producing a document in a software project. The Agile approach is a perfect example of choosing action over documentation (the agile manifesto labels it working software over comprehensive documentation) and often the documentation parts are ignored completely, even though that is a twisting of what Agile is meant to be. It is worth mentioning the RAD (Rapid Application Development) approach where

Some Reasons Why Projects Fail In The First Step (Requirements)

1. Failure to properly gather requirements
2. Failure to properly document requirements
3. Failure to communicate requirements
4. Failure to involve implementation knowledge

The Results of the failures:

1. We do not know what to build
2. We did not tell you what to build
3. We did not explain what we want
4. We did not realize our request was not feasible

The popular Agile methodology emphasizes "working software over comprehensive documentation"

requirements are not provided in a document, but instead are the application itself or some sort of proof of concept. This approach gets right into action (coding) and squeezes several of the steps together almost to the level of being implementation-deployment-maintenance in a single step.

Requirements gathering is not just about volume or time spent on the requirements. Many projects that fail before ever really getting started fall prey to spending too much time and effort on initial analysis and requirements gathering. This can lead into a trap known as "Analysis Paralysis" where too much time is spent on requirements gathering and the requirements step never really gets completed. The Agile methodology tries to address this situation by limiting the scope of the step and putting more emphasis on implementation. There are those in IT that are experienced and adept at producing requirements. The projects that succeed always seem to have those skilled resources involved early on in the cycle.

Design is the step that starts off where requirements gathering ends and the two steps can easily be blurred. If one considers requirements to be the end destination for a project, then design answers the question: "how do we get there?" The design is as important as the requirements gathering. When the requirements step is not done right, then the end product may not be useful to the customer. Bad design can lead to a product not useful to anyone. Design suffers some of the same lack of "wow factor" that requirements gathering does, as often the design step produces yet another document artifact (yada). This can lead to people thinking that requirements followed by design is just a "yada yada" before getting to the "real" work of implementation.

Good design is not the same as good implementation. The design step requires the resource(s) doing the design to understand the requirements and the business rules behind the requirements. Designers often grow out of the tech/implementation side and thus tend to have more advanced tech skills and less advanced (or sometimes absent) business knowledge. This technical bias often results in a "slick" technical solution to problems that do not need to be addressed, and the end customer getting a solution they did not ask for. Design is as much about filling in the gaps from the requirements step as it is about putting together a technical solution that meets the documented requirements. If you have ever done the exercise where you are asked to write instructions on how to tie a shoe you can relate to the difference

Some Reasons Why Projects Fail In The Second Step (Design)

1. Overly complex design
2. Design does not meet requirements
3. Design adds requirements
4. Failure to communicate design

Resulting in:

1. The cost is too great
2. It does not do what we asked
3. It is too difficult to use
4. We were not told what to build

Requirements can be seen as an answer to the question: "where are we going?" and design is the answer to: "how do we get there?"

Good developers are not always good designers and vice versa

between a requirement (the shoe lace should help the shoe fit snug and the lace not drag in a way that could cause a trip) and design (put each lace in a hand and …). In a software project a good design is a blueprint for the implementation. Completion of the design step often will result in a detailed design document and also often provides a roadmap or time frame for implementation milestones. Once requirements and design are done there is usually enough information available to build out an implementation project plan and to set target dates.

A design for a software project is also sometimes referred to as a blueprint.

There are experts at requirements gathering and also design experts. Design is a critical factor in the longevity and stability of a software product as it can effect implementation costs, maintenance costs, deployment, and many key factors in determining how "usable" a product is. It is a higher level skill than implementation as it must tie implementation to the customer needs (requirements) in order to be successful. This leads to design heavy positions typically requiring a greater number of years of experience and thus a higher price tag for resources.

High quality design can increase the cost of developing a product, but it will likely lead to a higher quality and more stable result.

Designers do not have to be the best implementors, but they do have to be able to communicate with the implementors down to a very detailed level. The best designers can work with implementors and verify that the implementors are "doing it right" by providing guidance, implementation suggestions, and even code reviews. The less experience designers have in the implementation approach used, the less effective they will be.

Designers are not the same as implementors, but they should be able to verify the design is followed.

Implementation is often the easiest step to understand. This is where the product is built. Code is produced, data stores are created, and all of the pieces are assembled. There are a lot of discussions/arguments about how much of a project is spent in implementation. The goal varies by methodology, but it is hard to find good numbers to determine how much time is spent on the steps for actual projects. It is not uncommon for implementation to consume 70+% of the total project time. Further reading on a particular approach will give an idea of how time should be apportioned to the steps when using that methodology, but most projects run based on target dates rather than percentage of time spent in a particular SDLC step.

Good developers are not always good designers and vice versa

It is not uncommon for the implementation portion of a project to take up 70% or more of the total time.

The implementation step is one that most often crosses boundaries with other steps. This goes back the mindset that producing code is the most valuable result of SDLC. Each

methodology has its own recommendations, but typically testing and deployment are done in close to parallel with implementation. Design is often done in part or almost entirely hand in hand with the implementation step (e.g. the aforementioned RAD approach). This allows design decisions to be proven or tested as they are made.

Implementation can include:
- Writing Code
- Test Script Creation
- Deployment Scripts
- Test Automation
- User Documentation
- Technical Documents

Implementation can be more than producing code. There are some methodologies that include the creation of test scripts, deployment scripts, the automation of tests and deployment, and even user documentation as parts of the implementation step as a whole.

It is worth noting that the implementation phase is often the one that has the largest number of resources assigned to it. Requirements gathering and design is often done by a small percentage of resources when compared to the implementation team. Requirements and Design for example, may be assigned to 2-3 people even when an implementation team might be 50 or more people. This equates to resources with implementation experience being far more prevalent than those that have a depth of requirements gathering or design experience. Designers will often be involved in implementation but in a team lead or management role to help leverage their unique skills.

The Implementation phase often has the most resources assigned and thus the most positions to fill.

Testing is arguably the least respected phase of the cycle. Over the last few decades, software development has progressed to where testing is seen as a necessary evil, but there are some companies that give it more respect. Testing is often built into one of the other steps as well as having its own period of focus in the process. Test planning is usually done in a way that allows for testing to be reduced or jettisoned in order to meet project deadlines. There is often an assumption that this reduction of testing will not have much impact on quality, however that is rarely the case. Testing recruitment is its own topic, but it is worthwhile to discuss some typical testing types and how they fit into SDLC.

Testing has not been seen as important in the past, but new standards and security concerns have made it a more common and visible part of SDLC.

- *Unit Testing* is the lowest level of testing and is intended to verify that a specific piece of functionality works correctly. An ATM machine would have unit tests to ensure that accounts are properly credited/debited, card reading works correctly, deposits can be read, etc. In a perfect world every minute function in a product would have validation unit tests that have been passed so we could be sure at least the most basic functionality pieces function correctly.

Unit Testing: testing small blocks of code

- *Integration Testing* is the next logical step from unit testing. Unit tests verify building blocks work individually and integration testing verifies the blocks work together correctly. In the ATM example an integration test might include a check deposit where the integration between verifying a check and the amount is integrated with making the proper credit on the account (and the account selection functionality).

Integration Testing: Testing how blocks of code work together.

- *Regression Testing* is a combination of unit and integration testing that systematically goes through all of the tests to ensure any recent changes to the implementation did not "break" the system. This is a testing type often missing, but highly desired, as it helps ensure application stability as code is enhanced or patched. Regression tests can be manual scripts but are often automated to some extent. It sometimes is called: "complete" testing.

Regression Testing: A repeatable process of testing where parts or the whole system can be "verified". Often this is automated

- *Smoke Testing* is a form of testing that focuses on the main functions of the application and touches each of them without going deep into any of them. This is often done immediately after a deployment as a form of "sanity check" of the production system. The testing goal is to catch any immediately visible issues that arose during deployment or only appear on the production system(s). This typically includes tasks such as: logging in, editing user information, clicking through the most commonly used screens and similar tasks.

Smoke Testing: A shortened test form to cover high visibility areas quickly.

- *System Testing* is sometimes used to describe Integration testing and might also be called UAT (User Acceptance Testing). This is testing that is done on the end product's ability to meet the requirements from an end user perspective. Functionality of the system is tested and rather than being a test of the implementation this is more a test of the requirements and design.

System Testing: Also called User Acceptance Testing is used to verify the end result meets the requirements.

The test names listed here are not set in stone and do vary from company to company and project to project. Testing types and how they fit into quality assurance as a whole will be covered in more detail in another IT4Recruiters white paper.

Test type names vary by company and vertical. Clarification of terms is always helpful.

Experience has shown that testers and implementors use a different skill set and they are not interchangeable. Developers tend to be very bad at testing their own code and few are adept at testing strategies. Developers can run test scripts, but it is rare for an implementation resource to be a good test writer.

Implementors and/or developers should not be used as testers!

In general, it is also bad practice to combine test and implementation roles into a single resource. Multiple people involved make it more likely logic flaws and incorrect assumptions are brought out in the testing process.

Deployment is a step that is overlooked at times because it tends to be the step that requires the shortest time and the least amount of resources to accomplish. The small time requirement can make it a task that gets bundled in with another role, such as an implementor or tester. This is changing as security and stability are becoming more important to almost every software project. There is an increase in complexity of deployments as more deployment targets (mobile, desktop, web) get included in typical releases.

Deployment: The step where the end result is made available to the users.

Deployment is typically a manual process early on in the life of a development shop, but as the team grows and improves its deployments it often moves to partial, and then complete, automation. Due to this advancement in how deployment is handled the deployment resource often begins as a part time task assigned to another team member, as noted earlier. As deployment moves to an automated process there tends to be a growth in need to a full time deployment resource that is either a converted team member or an additional resource added to the team.

Deployment often starts out as a manual process and a part time resource, but grows to automation and a full time job as complexity and maturity grows.

Deployment requires a blend of skills that includes the ability to understand how to “install” an application, system requirements, testing requirements, how source code is built into the application, and often include knowledge of system security. Senior specialists will have enough knowledge about the SDLC methodology being used to be a gatekeeper for when software is ready to be deployed. It is not uncommon for deployments in larger shops to include checking out source code, building/ compiling the source, running a series of automated tests, running database scripts, and deploying to multiple targets.

Deployment specialists often have a mix of skills including: coding, testing, network, and administration.

There is a form of development/deployment called continuous integration (CI) that makes deployment a part of the development and testing effort. This often results in nightly builds where developers commit changes for the day and a system based on the current code base is deployed for testing. A detailed discussion of CI is outside of the scope of this document, but is mentioned here as it does require a special kind of deployment resource to design and build the CI process..

Continuous Integration: Sometimes called CI, is a process where regular builds are made available (often nightly) and is a deployment specialization.

Maintenance is also known as "bug fixing" to most people. This step involves more than fixing bugs, but the only easily seen effect of a maintenance release tends to be bugs that are fixed rather than features that may be far more important, even if not easily seen by users. Unseen improvements often revolve around performance, stability, and logging system activity.

Maintenance: Best known as bug fixing or patching, this is where simple errors are fixed.

The amount of time and effort that goes into maintenance of a product is typically related to the product release cycle and how well the original release was implemented. The most notable exception to this is when third party products release an update that effects the product. This might include browser updates, operating system updates, and new versions of code libraries. A company that only does yearly releases is far more likely to have regular maintenance (also called point) releases than a company that does monthly releases. In the same way, a company that tends to release shoddy code on the first attempt will be more likely to spend time in maintenance releases.

Maintenance often includes performance and usability improvements as well as fixes.

The approach used to put together a maintenance release varies by methodology used but it typically is a form of shortened SDLC cycle:

- Requirements gathering is accomplished by gathering a list of known bug/issues
- Design is a minimal effort as little design is required to address bugs/issues in most cases
- Implementation is a short cycle due to limited amount of work/ changes to be done
- Testing may be minimal and targeted to the areas impacted. Regression testing is highly useful for maintenance releases
- Deployment is the same process as used to do a full release in some cases although it is becoming more popular to do a "patch" that only changes/updates some of an existing deployment

It is not uncommon for maintenance releases to follow a shorter, but full SDLC life cycle. The Agile approach has a concept of "sprints" which effectively makes all releases a shortened SDLC cycle.

Maintenance is often considered a less challenging task than new development and might be assigned to a smaller/less experienced team than the original development team. This is also work that some find to be "beneath" them and that can lead to maintenance roles being shunned. Unfortunately none of this reduces the importance of maintenance in software stability.

Maintenance lacks glamor as a role, but it is critical to on-going success for a software product.

Adoption

SDLC is here to stay in some form and it is steadily evolving. There are still startups and first time development attempts that do not follow some form of SDLC approach, but even that is becoming less common. Proper understanding of and the ability to adhere to a SLDC methodology is still an issue for many teams. It is not uncommon to find a software development team that has taken a methodology or two and adopted them to the team's strengths, weaknesses, goals, and comfort. Design and testing often start out as less a part of the total effort. As a team or company matures, implementation shrinks as a percentage of the whole effort while testing grows to a more important role. This recognition of the import of design and testing leads to older, more established companies not only likely to have more ingrained SDLC processes, they are also more likely to be looking for resources experienced in a particular methodology.

SDLC has been widely adopted and is steadily evolving and advancing. This is an aspect of IT that is key to understanding the ebbs and flows of the hiring "seasons".

Strengths, Weaknesses, Progression

SDLC is often a secondary or tertiary trait that will show up on a list of position requirements. This makes SDLC experience that matches a hiring company more often a "plus" or "nice to have", than a hard requirement. The exceptions to this are: companies that are trying to improve their adherence to a methodology, that want experienced resources to lead that effort, and agile projects. Agile is considered a new approach and is a very different way of thinking when compared to the older waterfall. Companies often want resources that have been exposed to agile to avoid difficulties in integrating a new member into the team due to lack of understanding of the methodology.

SDLC experience is a factor in hiring decisions and asked about in interviews far more often than it appears on the list of job requirements.

In the early stages a company will be searching for implementation resources almost exclusively. The design and requirements gathering will be done by the implementation team or a team lead. As the company matures there will be design resources added. Testing is often grown by requests for implementation resources that have some sort of testing/QA exposure and/or experience that moves into requests for skilled and experienced quality resources. These requests often tend to arise after a failed or low quality deployment. Deployment specialists are added towards the end of building a "complete" team and often after a few product releases. It is often the case that a company suffers due

Companies tend to start out hiring implementation resources and then move to designers, analysts and eventually testers, as they mature in using SDLC.

to bad testing or a botched deployment and then starts a search for QA or deployment resources. It can be very valuable to a relationship with a customer to be ahead of the game and suggesting resources within the customer's budget before a QA or deployment disaster hits.

CMM: Capability Maturity Model is a measure of the level of adoption of SDLC (among other things) and mention of it on a job description should be considered an indicator that SDLC experience will be addressed in screening and interviews.

Market skill set

In general, it is hard to spend any time in software development and not be exposed to SDLC of some sort. It is a very academic subject in many ways and taught in most computer science programs. A number of IT related general studies will touch upon SDLC at least as an overview. Methodologies are often topics covered on tech blogs and in tech magazine articles, so exposure at some level often occurs early in an IT career. Jobs at large corporations often will include SDLC exposure although that exposure tends to be focused on a single step and not the full life cycle.

Experience is the best teacher of SDLC and it requires full life cycle exposure. This leads to SDLC skills that are requested often starting with a minimum of 2+ years of experience and "senior" resources can easily require 15, 20, or more years of experience. This varies by methodology, as some are very new. For the newer methodologies, experience in a single project using that methodology can be considered strong experience. When in doubt, try to find out how old a methodology is when trying to meet job requirements, it will help avoid trying to fill a position for someone with 20 years experience in a five year old technology.

SDLC Skills are most often based on experience more than any certifications or formal education as that is the best teacher.

Certifications

SDLC in general does not have a certification although almost every methodology has some form of certification available. There are links at the end of this document to help and a Google search of the methodology and certification is often fruitful with good matches at the top of the results and the search sponsors/ads tend to be a good source of further information.

There is not a general SDLC certification, but you can find them for specific methods and Agile in particular.

SDLC Conversations

Early in this document it was pointed out that SDLC tends to be a secondary requirement at best; however, this does not make it something that should be ignored when trying to fill a position. It is important to have an understanding of whether a resource will need to be able to meld with the team (and their approach) on day one, or whether a new team member will be able to learn the methodology on the job. The lack of SDLC appearing on a job description should not preclude it as part of the discussion about best fit resources.

SDLC is important enough to always make it a part of what sort of resources are a best fit for a position.

Clarifying Questions

Whether you are trying to define a position to be filled or are trying to find resources that are good candidates to fill a position, there are a few things that are helpful to know when making that match:

- What methodology, if any, is used?
- How important is SDLC methodology to the role being filled? Is this a company that sort of uses the chosen methodology or adheres to it strictly?
- Does the organization have mentors for the methodology? are they hoping to find mentors?
- Will the candidate need to know the whole SDLC for the methodology or just some of the steps? Which steps?
- What is the SDLC maturity level of the company? Is this their first release or have they been following the chosen methodology for years?
- What is the typical release cycle? Larger projects and longer release cycles mean longer and more involved SDLC steps. A resource may be a better fit if they have more experience at the point of the life cycle where the company is at when the resource joins the team.

SDLC Questions:

- What methodology is being used?
- How important is SDLC to the role?
- Are mentors available?
- Is full SDLC knowledge needed?
- What is your maturity level for SDLC?
- What is the typical release cycle?
- Where are you at in the release cycle?

Crossover/Complementary Skills

The list of complementary (or replacement) skills for an SDLC requirement is short due to how general SDLC is as a topic. There are similarities among the various methodologies, so you can find candidates that are going to be solid in an agile shop due to their waterfall experience, for example. This is not always the case, so the only way to get a good feel for a skill set that is going to be "close enough" to SDLC requirements for a position is to get to the root of why any SDLC requirements are listed on the position. The hiring manager may just want some SDLC experience as a way of proof that a candidate has been involved in a "real" project development process, or there may be a specific gap in the current team knowledge that needs to be filled. As was noted in the prior section, SDLC should always be discussed, but if it is already a part of the job requirements that is (obviously) a great starting point for the discussion.

SDLC crossover skills tend to be from other methodologies, but still within the general sphere of SDLC.

Weasels

SDLC weasels generally fall into a couple of categories. They either try to convince the customer that they have full SDLC experience when they do not, or they try to show experience they do not have in a specific step or steps. In both cases, weasels tend to use terms and buzz words they do not understand, but they count on the listener to be impressed. They think that simply using the words will show how knowledgeable they are. This makes it fairly easy to spot the weasels by asking them to explain the buzzwords they use. A weasel will often avoid specifics in their explanation/definition (or might be completely wrong) where a truly experienced candidate will provide a specific definition and be able to provide examples from their experience.

When in doubt: Ask for further explanation of terms and buzzwords. This is the best way to show weasels for what they are.

SDLC is a technical skill set that has produced a lot of candidates that have less knowledge than they realize about SDLC in general and methodologies in specific. Agile is the worst case of this as it is considered a "hot" skill to have, but many candidates that use Agile miss the point and only have limited experience with the full methodology. There is a lot of terminology around Agile (scrum, pair programming, xp, extreme, sprint, etc.) that gets thrown around without knowledge of what the terms actually mean or practical experience. As with any methodology, it is best to spend some time learning the key terms used by the methodology and what they mean. In particular, what they mean to the hiring

Agile related terms are hot topics in the current market so a general understanding of these terms can be highly valuable in avoiding weasels.

manager as the manager may have a different understanding from the "official" definition.

Screening Questions

Here is a sample list of questions to help screen a candidate based on their SDLC experience. These may come up in discussions about the job description and/or may be asked in a typical technical screen/interview:

- What are the components/steps of SDLC? Terminology varies so you may need to ask for a description of a term to determine whether the answer is correct.
- What are the outputs of the SDLC steps? This can be modified to ask about a specific step if knowledge of the whole SDLC is not required.
- Have you been involved in a project that went through an entire SDLC cycle?
- What SDLC steps are you most comfortable with and why? This will help determine the kind of experience the candidate has and their natural inclinations along with getting a further idea of their knowledge of the step(s) they choose.
- Is there a particular SDLC methodology you prefer or are most comfortable with? Why? This is a more advanced question, but may be very important when specific methodology is required or preferred.

SDLC is such a common facet of software development that more questions and ways to test for SDLC knowledge are available. These questions are a good start though. Rather than look for a specific answer to these questions, let the candidate talk as that may provide far more insight into their knowledge and experience.

Good SDLC screening questions are typically open ended and intended to lead to the candidate giving long form answers rather than a simple, or canned answer.

SDLC Resources

Resources for more information

http://istqbexamcertification.com - Lots of information is available at this site if you want to dig deeper into this topic and several related topics.

http://www.aspe-sdlc.com is a training and certification site that has a few free resources and a number of training offerings to learn more about many different SDLC approaches including the popular Agile and Waterfall approaches.

http://www.pmi.org is the site for the PMP certification which is Waterfall oriented and a proponent of that approach.

http://www.pmi.org/Certification/New-PMI-Agile-Certification.aspx Is a new certification pmi.org has added for agile approaches making pmi.org a great place to compare and contrast waterfall and agile approaches to the SDLC.

Source Materials

http://istqbexamcertification.com - Lots of information is available at this site if you want to dig deeper into this topic and several related topics.

Salary info for quick reference:
http://www.glassdoor.com/Salaries/business-analyst-salary-SRCH_KO0,16.htm
http://www.payscale.com/research/US/Job=Business_Analyst,_IT/Salary
http://www.payscale.com/research/US/Job=Software_Designer/Salary
http://www.payscale.com/research/US/Job=Software_Developer/Salary
http://www.payscale.com/research/US/Job=Quality_Assurance_Analyst/Salary/4763392b/Late-Career