Chapter 2
Information System Development Life Cycle Models

2.1 Introduction

With the increasing evolution and complexity of information technologies, there has emerged a multiplicity of applications for information systems (IS): They assist in corporate transactions, they connect business and office data, and they support users in the architecture of strategy. The complexity of their nature and objectives requires the harnessing of technology and user experience to create systems that meet their expected purpose. In essence, system development consists of this process of creating an information system, with all the variables that it entails and which usually need to be taken into account: its ability to be user-friendly, how well it functions, if it meets the needs of the organization in which it will be integrated, and so forth.

The life cycle of an IS development begins with its creation and ends with its termination. Along this process, it goes through various stages, which have been discussed to some extension in existing literature. Cohen (2010) outlines "requirements, analysis, design, construction (or coding), testing (validation), installation, operation, maintenance, and the less emphasized retirement" as the key components of the development process. According to Jirava (2004), the conventional "life cycle is composed of five phases: Investigation, User Requirements, Analysis, Design, Implementation and Release."

Generally speaking, the life cycle is perceived as the time frame that spans from the development of a new system to its eventual retirement. It is a process that starts with the emergence of an idea, goes through its implementation, and ends with its termination, moving across all the intermediate stages in which its viability and usability are prioritized (Jirava 2004).

However, IS are very complex structures. They are built with a specific goal, for a specific organization. Because of this specificity, each system development process requires a guiding framework to configure, outline, and monitor the progress of the development along all the stages of the life cycle. Although the methods employed in this framework depend on the peculiar characteristics of each project,

P. Isaias and T. Issa, *High Level Models and Methodologies for Information Systems*,
DOI 10.1007/978-1-4614-9254-2_2

it is possible to ascertain that there are key components that all congruent frameworks must necessarily entail. The most notable one is the segmentation of the development process into phases, with each phase having a beginning, an end, a series of specific activities, deliverables (documentation that is prepared regularly to ensure performance accountability for each required task), and monitoring tools. Cohen (2010) notes that this same principle is also true for the introduction of IS not through their development in house, but through external acquisition (the purchase of a set of software applications from an external vendor). Both imply a process of implementation, maturing, and termination.

The core objective of the development or implementation of a system is its efficient integration in real-life situations. Therefore, two of the primordial steps in the life cycle are the assessment of what the different people involved in the system's use will require and a knowledge of the context in which the system will be operated. The negligence of these two essential elements is at the origin of several issues in systems' use in real settings (Tetlay and John 2009). The usability aspect of a system is central, that is why the inclusion of the user in the entirety of the development life cycle is crucial. For developers, it is paramount to have all the correct information regarding the users' needs. Misinformation has repercussions on the development of the system, and it usually results in the implementation of products that fall short of the users' expectations and have a diminished productivity. It is important to potentiate the users' participation, namely by informing them about how the development process works and the need for accurate information (Durrani and Qureshi 2012).

However, even considering these crucial common points that all frameworks must necessarily include, the methodology for system development has a great variety of approaches, which we call models of system development life cycle (SDLC). Some of the most commonly cited are the spiral, waterfall, V-shaped, and agile models. Given the multiplicity of research studies on this field, it is crucial to outline an overview of them, to help further understand which one of them would be more appropriate for a specific project. The point where the model or framework that will guide the development process is chosen is a central strategic aspect that will undeniably have an impact on the effectiveness of the system in the long run. The wrong life cycle can delay the project and affect client satisfaction, and it can even mean the cancelation of the system (Executive Brief 2008).

In this paper, we will offer a panoramic overview of the most significant models of SDLC, which will present a useful tool for the embryonic stages of any IS development process.

2.2 The Waterfall Model

The waterfall model was introduced by Royce in (1970), specifically in the context of spacecraft mission software design, and is one of the most popular methods of assessing the evolution of a product or system. Essentially, it is a step-by-step

sequential description of the product's life cycle that spans 7 different stages, originally denominated "system requirements, software requirements, analysis, program design, coding, testing and operations" (Royce 1970).

The first premise in which this model is based is that any development process of any software or system starts off by two essential steps: analysis and coding. This is the simplest conceptualization of the model, but is ineffective to understand the product's further development beyond the stage of creation. Therefore, the analysis stage is broken down into two steps—analysis of both system and software requirements, while the coding stage is preceded by program design (Royce 1970).

The essence of the waterfall model is that it attempts to provide a useful set of guidelines for the development of new programs or systems. In his original work, Royce (1970) provides five key principles that he believes are essential for the successful development of large software systems.

The first is "program design comes first." It is essential to allow designers to be a part of the initial process, because of their invaluable feedback regarding resources and limitations. The second is "document the design." Extensive documentation of the development process is paramount, not just to facilitate management of the process, but to facilitate performance assessments, making the eventual correction of mistakes more efficient. The third is "do it twice," referring that the final version of the product should actually be the second version, where all the stages have been performed and it is easier to pinpoint strengths and weaknesses, to emphasize the first and correct the latter. The fourth is "plan, control and monitor testing." Testing is a fundamental stage. It is important to bring in specialists that did not participate in the earlier stages of the process. It is also important to test every single aspect of the project, regardless of how relevant it is. Finally, the fifth guideline is "involve the customer." Having the insight, judgment, and commitment of the customer taken into account during the development process is a viable option that will greatly improve its potential for general acceptance (Royce 1970).

The waterfall model was a popular approach, and for that reason, it evolved and adapted into numerous forms, according to different research studies and the context of application. Denomination of each step varies greatly and can reflect the specific objective of the study or the field in which it is applied.

However, one common trait covers all the variations of this model: it is a sequential model. Each of its stages must be entirely concluded before the next can begin. Similarly to the flow of a waterfall, the development of the software is regarded as continuously streaming downward throughout its different stages (Massey and Satao 2012). Thus, for example, analysis of requirements must be thorough and final before design begins, and testing can only be efficiently carried out once coding is entirely complete. Each stage is regarded as a static component, a rigid step in the process. Subsequent changes in previous steps (e.g., awareness of new requirements) cannot be taken into account (Balaji and Murugaiyan 2012) (Fig. 2.1).

The waterfall model took precedence over other models in the 1980s and the beginning of the 1990s. But this preponderance suffered an important setback with the increasing speed of technological evolution and the subsequent need to swiftly

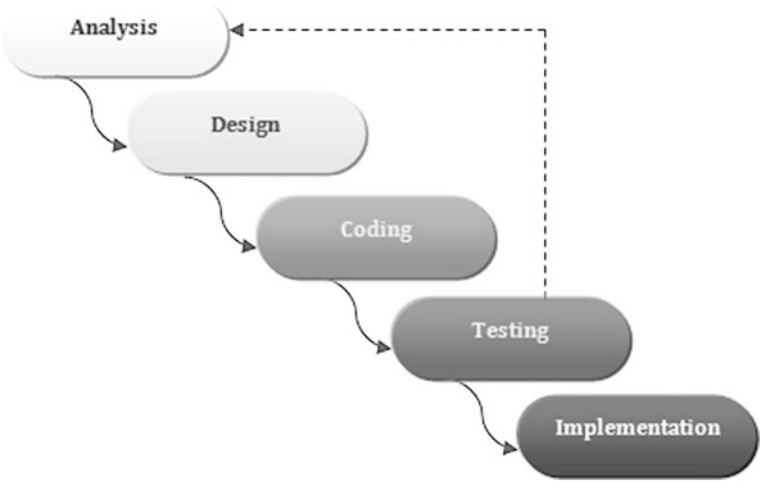

Fig. 2.1 An example of the waterfall life cycle model (adapted from Balaji and Murugaiyan 2012)

deliver new software systems and products. Viewing each stage as a single, "frozen" moment of evolution can greatly delay the implementation stage because errors will only be detected very late in the process, during the testing phase, which is preceded by extensive designing and coding (Munassar and Govardhan 2010). The communication of objectives between developers and clients is also greatly hindered because if the client changes the requirements of the system, the development process needs to completely restart for those changes to be taken into account (Balaji and Murugaiyan 2012).

This model is an idealized and greatly simplified concept of SDLC. It is not very flexible, but it is still popular as a conceptual basis for other frameworks or models. Its greatest strength lies in that it outlines generally accepted positive habits of software development, such as minute and accurate planning early in the project, extensive documentation of the entire process, and having robust design concepts before starting to code (Munassar and Govardhan 2010). However, the reality of a development process can often be much more disorganized than that.

2.3 The Incremental Model

The incremental model is a particular evolution of the waterfall model that attempts to address its more prominent shortcoming, which is the slowness of the cycle. It also aims at outlining a more flexible process that requires less extensive planning up-front (Munassar and Govardhan 2010).

According to this approach, instead of dividing the SDLC into static, isolated steps, the whole process can instead be designed, tested, and implemented one

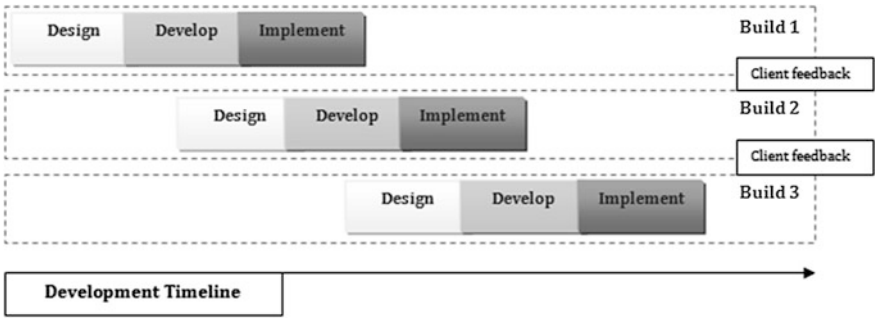

Fig. 2.2 An example of the incremental life cycle model (adapted from Tilloo 2013)

fraction at a time, in successive stages, so that with each stage (or increment), there can be at least some feedback from the client. This feedback will provide valuable assistance in the next increment of the process and so forth. With each ongoing increment, the product is extensively tested and improved, according to objectives and expectations from the client, which facilitates its eventual success (Massey and Satao 2012) (Fig. 2.2).

With this SDLC model, the process of software development is made by increments, through a series of different releases. When the product is launched, when it has its first increment, it is ready for consumer use. Then, according to the clients' response to the software, new increments are made to improve the product. The increments will continue to be added, until the completion of the final product (Massey and Satao 2012).

Each stage is scheduled and structured to allow the development of parts of the system at varied rates and times and to incorporate them into the global project when they are finished. Thus, this model highlights the sequential process of the different phases of development while also trying to maximize the benefits of allowing changes, improvements, and additions to be made between each increment. Development is broken into smaller efforts. These are consistently monitored, so that progress can be accompanied and measured (Texas Project Delivery Framework 2008).

This model fundamentally outlines a progressive development process through the gradual addition of more features, until the completion of the system. It is a more flexible method, because it allows for the incorporation of needs that might not have been obvious at the start of the process, and it facilitates the changes that come with later assessment of different requirements. Additionally, since it builds on each of the phases, it allows for wider amplitude of improvement in the following stages (Executive Brief 2008). Thus, product delivery is not only faster, but it is also easier to test and eventually correct.

However, the downside to this approach is that it can be more costly to develop and release multiple versions of the product. On the other hand, when a later increment is developed due to a new found problem or necessity, it can have compatibility issues with earlier versions of the product (Tilloo 2013).

There is a variation of the incremental model, named the iterative and incremental development model (IIDM). While very similar to incremental, this model puts greater emphasis on the relationships that occur with each increment and between them. These relationships, or iterations, form a cycle or pattern of feedbacks and outputs. In that regard, while maintaining the essence of the incremental model, the IIDM is a more fluid description of development. At the same time, it also allows more space and significance for feedback, as it modifies the scheduling strategy to include specific time frames for revision and improvement of each increment, so that successful conclusion of the development process is more likely at the first final version (Cockburn 2008).

2.4 The Spiral Life Cycle Model

The spiral model dates back to the end of the 1980s, when it was outlined by Barry Boehm, and introduces something that other models did not take into account, which is risk analysis. In essence, the spiral model attempts to bring together key aspects of some other prominent models (namely the waterfall, incremental, and evolutionary prototyping), in an attempt to gather the most appropriate traits from each one, because specific projects might be more or less adaptable to specific models.

According to this SDLC model, the process of developing a system consists of a series of cycles or iterations. Each cycle begins with the identification of objectives and requirements of the current stage, as well as an analysis of alternatives and constraints. This process will highlight areas of uncertainty (risk), which will be taken into account during the next step, the outlining of a strategy or plan, through prototyping and other simulation methods. This process involves constant improvement of the prototype as risks are decreased (while others may arise). Once the prototype becomes sufficiently robust, and risk is reduced to acceptable levels, the next step develops in accordance with the basic waterfall approach, through a succession of stages: concept, requirements, design, and implementation (Boehm 1988). Once this cycle is concluded, another cycle begins, as a new increment of the product is created.

The spiral model bears some resemblance to the incremental life cycle, but the emphasis on risk evaluation presents a major difference. The stages or spirals that constitute this model regard planning as a first step, moving then to the exploration of what the requirements are and subsequently calculating the risks. In this stage of risk calculation, the model is structured to initiate a process of determination of risks and of formulation of alternatives (Massey and Satao 2012); thus, risk management can be considered the centerpiece of the model (Fig. 2.3).

Boehm (2000) asserts that each cycle or iteration of the process will invariably display six particular characteristics, which he named the “invariants.”

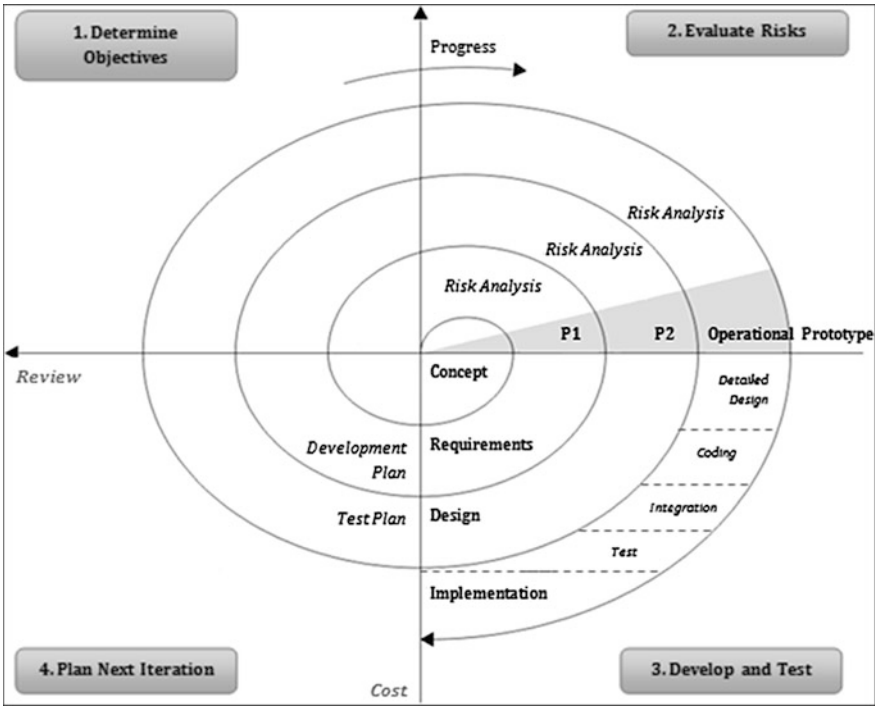

Fig. 2.3 A summary of the spiral development model (adapted from Boehm 1988)

Invariant 1 is the *concurrent definition of key artifacts*, such as concept, requirements, plan, design, and coding. It is argued that defining these artifacts in a sequential structure can constraint the project to excessively rigid preconceptions.

Invariant 2 is that *each cycle follows the four strategic principles that correspond to the four quadrants of the model*: determine objectives, evaluate risks, develop and test, and plan the next iteration. Not moving in accordance with this basic strategy can negatively affect the entire process.

Invariant 3 is that *the level of effort is determined by the risk considerations*. Reasonable time frames must be established for each project in accordance with risk assessments, to determine "how much is enough" in each activity.

Invariant 4 is that *the degree of detail is driven by risk considerations*. Just like invariant 3, here it is important to determine "how much detail is enough" in each stage of the process.

Invariant 5 refers to *the use of anchor point milestones*, which Boehm describes as "Life Cycle Objectives (LCO), Life Cycle Architecture (LCA) and Initial Operational Capability (IOC)" (Boehm 2000). At each of the anchor points, stakeholders will review the key artifacts of the stage.

Finally, invariant 6 states that besides the construction aspects, *the development process needs to focus also on the overall life cycle itself*. This means that long-term concerns should always be taken into account.

The spiral model has significant advantages over previously described models. The emphasis on risk analysis provides a major improvement and makes it an ideal model for large, mission-critical projects (Munassar and Govardhan 2010). On the downside, it is not very efficient in smaller projects; the risk assessment process can increase the expenses of the system to a degree where even making the system, regardless of risks, can be more financially sustainable. The risk assessment is also a procedure that demands a very peculiar expertise and needs to be custom-made for every system, which will contribute even further to a steep rise in costs (Rahmany 2012).

2.5 The V Life Cycle Model

The V-Model was presented in the final years of the 1980s by Paul Rook, as a variation over the waterfall model that attempted to emphasize the existing connection between each of the stages of the development process and its respective stage of tests. By focusing on this relationship, it ensures that adequate quality measurements and testing are constantly resorted to throughout the life cycle (Skidmore 2006).

The method thus presented is that each step is implemented by resorting to detailed documentation from the previous step. With this documentation, the product is checked and approved at each stage of the process, before it can move on to the next stage (Balaji and Murugaiyan 2012). With constant testing, and its respective documentation, it is possible to increase the overall efficiency of the process, particularly because eventual problems can be detected and resolved early (Mathur and Malik 2010) (Fig. 2.4).

The V-Model starts off with a very similar premise to the classic waterfall models. In successive steps, the project goes from analysis of requirements and specifications, to architectural and detailed design, to coding. However, instead of

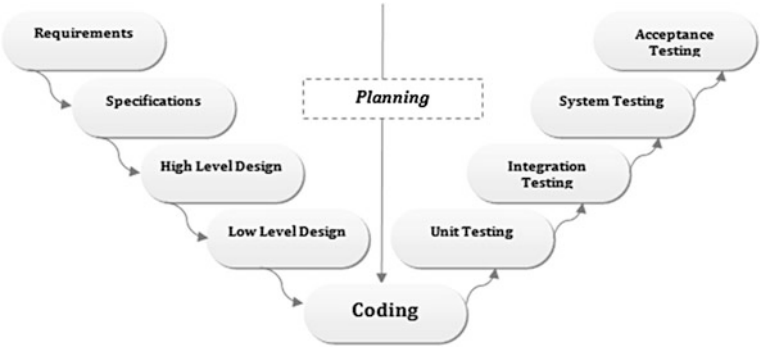

Fig. 2.4 V life cycle model (adapted from Balaji and Murugaiyan 2012)

continuing this downward ladder, there is a parallel structure that moves upward from the coding stage, giving the model its distinct V shape. The upward ladder describes each of the testing steps that follows coding, starting with unit testing and ending with acceptance testing, the final step before final release (Mathur and Malik 2010).

In that sense, the V-Model describes three successive layers of system development that can be described as requirements (overall system), high-level design (system architecture), and low-level design (software components). To each of these layers, there is a corresponding layer of planning and testing. Planning is, indeed, the axis that stands between the left and right ladders that compose the V, as it is the mediating action set between design and testing (Munassar and Govardhan 2010).

The core objective of the V life cycle model is to illustrate the importance of the relationship between development and testing tasks. Nonetheless, the success of a project is also determined by its maintenance structure. To accommodate this reality, Mathur and Malik (2010) proposed an expanded version of the V-Model, called the advanced V-Model, that incorporates both testing and maintenance activities into the life cycle. This adds a third structure, or "branch," to the model that reflects the introduction of testing mechanisms after the final product is released, so that proper quality measurement, troubleshooting, and general maintenance can be ensured.

Since the V-Model addresses its errors shortly after they are identified, it becomes less expensive to resolve them, which is perhaps the greatest advantage of using this model, specifically when compared to the classic waterfall model. Also, because testing is fractioned throughout the process, all the parties of the development are responsible for it. This also means that testing methods are adequate to each of the stages. Furthermore, the fact that tests are performed since the beginning of the process only increases its efficiency (Mathur and Malik 2010).

On the other hand, and much like the waterfall model, this model is very rigid and there is little room for flexible adaptation, particularly because any alteration in the requirements will render all existing documentation and testing obsolete. Since it requires a great deal of resources, it is clearly optimized for large projects within large organizations (Rahmany 2012).

2.6 Rapid Application Development

Originally conceptualized in the 1970s, the rapid application development model was substantially developed and formalized by James Martin in the early 1990s. As the name suggests, it is driven by the idea that existing life cycle models are simply too rigid to permit a fast project development; therefore, there is need for a framework that can account for fast delivery while still maintaining high-quality standards. It is grounded on the principle that step-by-step structured life cycles inevitably entail delays and errors, urging the need for an alternative methodology.

This issue became more relevant as businesses became increasingly competitive and IT needed to keep up. When deadlines are the main priority and the swiftness of software development is critical, RAD presents itself as a very plausible solution.

RAD comprises a set of tools and guidelines that facilitate short-time deployment, within a predefined time frame or "timebox." The product is not developed in successive steps until a final, complete delivery, but rather it evolves in successive increments, following the priorities that are established by business—not technical—necessities (Gottesdiener 1995). Some of these tools and guidelines include planning methods, data and process modeling, code generation, testing, and debugging (Agarwal et al. 2000).

It is important to note that both developers and customers are involved in all of the increments. However, teams are generally small, highly skilled, and highly disciplined. They are required to flexibly adapt to eventual changing requirements and feedback from customers. Nevertheless, it is crucial to strike the proper balance between flexibility and structural stability. Underlying models to the product's design are still necessary (Gottesdiener 1995), but not as rigid step-by-step guides to be followed to the letter (Fig. 2.5).

RAD methodologies can follow three-stage or four-stage cycles. "The four-stage cycle consists of requirements planning, user design, construction, and cutover, while in the three-stage cycle, requirements planning and user design are consolidated into one iterative activity" (Agarwal et al. 2000).

During planning, it is possible to analyze requirements, alternatives, and opportunities, as well as possible risks. This will form the basis for a definition of the project's goals and scope, and more importantly, it will allow for the establishment of the timebox, which is a fixed period during which a specific increment of the product is going to be developed. Each increment is then developed in a spiral-like model, through design, prototyping, and testing. This method essentially pushes the team closer to the project's business goals, by providing key deadlines that can be determined by market forces (Gottesdiener 1995).

While the advantages of the RAD are evident, due to its focus on swift delivery and effective developer–client communication, there are still a number of issues raised by this approach. One of the most obvious flaws is that it removes a great

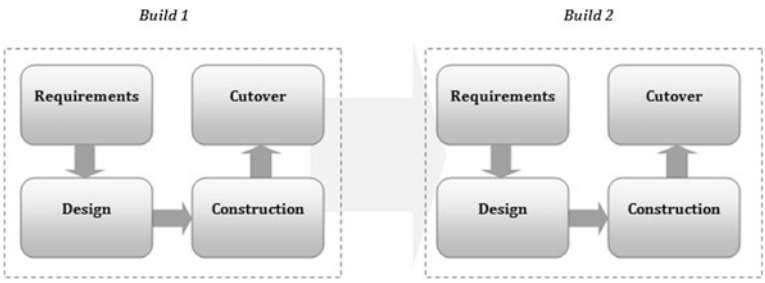

Fig. 2.5 Rapid application development entails a succession of increments or versions of the product, each built on a predetermined timebox and following a cycle of 4 (or 3) steps

deal of emphasis on minute planning and modeling at the start of the project, shifting that focus to the fluid process of system construction (Agarwal et al. 2000). Another prominent issue is that in faster development cycles, extensive quality testing will become less prioritized, reflecting in poorer quality overall, which means that effective RAD methodologies should reserve space for skilled individuals in quality control roles (Gottesdiener 1995). It is also possible that managers and leader have unrealistic expectations regarding the timeboxes, creating conflict with developing teams (Agarwal et al. 2000).

Thus, it is possible to assert that in order to be optimized, RAD life cycles must necessarily be balanced and be open to moderating agents.

2.7 Agile Life Cycle Model

With the popularization of waterfall-like SDLC models, an alternative approach has been developing that attempts to counter their rigidness and lack of flexibility. We have seen such examples in the incremental and RAD models. In 2001, the manifesto for agile software development was presented by 17 software developers, in a new attempt to bring together the best traits of other agile-like models into one framework. Since then, agile methods of development have become increasingly popular (Bhalerao et al. 2009).

There are 12 principles that guide agile development models, which were outlined in the Agile manifesto. These principles can be summed up as follows (Beck et al. 2001):

- Customer satisfaction is the highest priority;
- Change in requirements is welcomed, no longer an obstacle;
- Software is delivered regularly in consecutive releases;
- Motivated individuals are key to successful projects;
- Face-to-face conversation is paramount to successful collaboration;
- Working software is the measure of the project's progress;
- Sustainable development should be encouraged;
- Emphasis on technical and design quality;
- Simplicity should be favored;
- Self-organizing teams are the best form of project development;
- There should be regular discussions on team improvement.

There are numerous subvariations of the Agile model that follow these principles, with some examples being the scrum and XP models. However, even considering the variation in timescales or stage description, it is possible to determine the general path that an agile development process will take, outlined in four steps.

The first step is *project selection and approval*. During this stage, a team consisting of developers, managers, and customers establishes the scope, purpose, and requirements of the product. There is also a thorough analysis of different

alternatives to accomplish the established goals, as well as risk assessment for each idea (Bhalerao et al. 2009).

The second step is *project initiation*. After the establishment of a coherent project with respective goals and scope, a working team is built, with the appropriate environment and tools, as well as the working architecture in which the system will be based. This too is discussed among all stakeholders. At this point, it is also adequate to establish working time frames and schedules (Ambler 2009).

The third step is *construction iterations*, with each iteration consisting of both planning and building. Developers release working software in successive increments that will accommodate the evolution of requirements as outlined by the various stakeholders. Close collaboration is therefore a fundamental aspect of this process, as the most effective method to ensure quality and to keep the project's priorities well defined. Extensive testing of each iteration is also paramount at this point (Ambler 2009).

The fourth and final step is *product release*. This stage encompasses two stages: First, final testing of the entire system is done, as well as any necessary final reworks and documentations. Next, the product is released, at which point training is provided to the users in order to maximize operational integration. The working team might maintain the project so as to allow for product improvement as well as user support (Bhalerao et al. 2009) (Fig. 2.6).

The Agile SDLC emerged from the ever-increasing need to match the speed at which IT evolves. What sets it apart is its dexterity in developing products at a great speed, with products being deliverable in the course of weeks instead of months. This is possible due to the model's emphasis on collaborative efforts and documentation (Executive Brief 2008).

Another advantage of the Agile model is that it is very flexible. It has been occasionally combined with other existing models. It has the capacity to deliver systems whose requirements go through constant changes while, at the same time, demanding strict time limits.

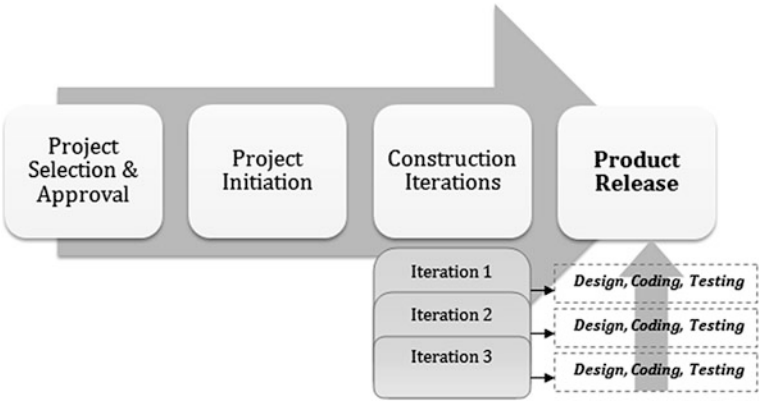

Fig. 2.6 An example of an Agile development life cycle

Finally, this model is often praised for its high degree of client satisfaction and user-friendliness, reduced error margins and the ability to incorporate solutions to address the needs of highly mutable requirements. Agile models are client-centric and advocate "short iterations and small releases" in order to obtain feedback on what has been accomplished. With the feedback that is received, improvements can be made that will have positive repercussions on the quality of the end product (Bhalerao et al. 2009).

2.8 The Prototyping Model

The prototyping model is an iterative framework that is at the center of many of the more agile approaches to software development, ever since the early 1980s, which lead to it being described in some studies as a specific model in itself. In 1997, Carr and Verner observed that in the past research, the SDLC models that adopted prototyping were found to be more dynamic and more responsive to client needs, as well as less risky and more efficient. For that reason, they attempted to summarize prototyping models in one consistent framework.

The prototyping model is based on the idea of creating the entirety or part of a system in a pilot version, called the prototype. It can be viewed as a process, either one that is part of the larger SDLC or the central approach that defines the SDLC in itself. The goal is ultimately to build in various versions and consistently refine those versions until a final product is reached (Carr and Verner 1997). The emphasis is placed on the creation of the software, with less attention to documentation. It is also a user-centric approach, because user feedback is fundamental to develop subsequent prototypes and, eventually, the final product (Sabale and Dani 2012) (Fig. 2.7).

A prototyping model essentially entails four different stages. First, user's requirements and needs are analyzed and identified. Next, the team will develop a

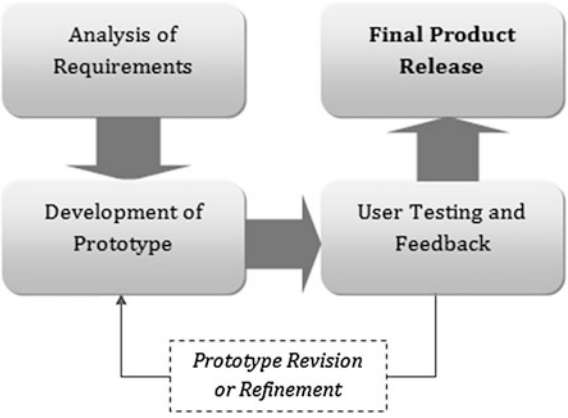

Fig. 2.7 The prototype model (adapted from Carr and Verner 1997)

working prototype of the product, which is then implemented so that the users can test it and provide real-time feedback and experience. If improvements and changes are found necessary, the prototype is revised and refined, and a new prototype is released and implemented for testing. This subcycle will go on until the product is generally accepted by the users and no longer requires substantial changes or updates, at which time the final version is released (Carr and Verner 1997).

There are various types of prototyping, according to specific needs of the project. These can be summarized in three categories: exploration, experimentation, and evolution (Floyd 1984).

The *exploratory approach* is centered on the premise that requirements are thoroughly explored with each iteration. Under this category, we find rapid throwaway prototyping, essentially a method of delivering fast releases of the product with each iteration, exploring needs and requirements with each version, and perfecting the next version accordingly. Needs are assessed as the product is used and tested. On the other hand, the spiral model, which we have previously discussed, is another form of exploratory prototyping, where prototypes are employed in successive stages of the development process, each following the waterfall pattern (Carr and Verner 1997).

The *experimental approach* entails that a solution to the user's needs is first proposed and then evaluated through experimental use. The use of simulation programs and skeleton programming (delivering only the most essential features of the system so the user can get a general idea of what the final product will be like) falls under this category, but there are many other examples, as it is the most common form of prototyping (Floyd 1984).

Finally, the *evolutionary approach* essentially describes development in successive versions and is closest to incremental and iterative life cycle models, in that its main goal is to accommodate the eventual changes in requirements and needs. The prototype is used fundamentally to allow for easier contact with the product, in order to pinpoint perceived needs. Each prototype is no more than a version of the product, and each version serves as the prototype for the next one (Floyd 1984).

By using a form of the prototyping model, a development project can easily adapt to changing requirements, because there is constant feedback. With each iteration, or version of the product, the user will have the ability to test the prototype and provide valuable input on its traits and requirements. This provides the model with much higher probabilities of success, as well as low risks. On the other hand, because there is not much emphasis on extensive documentation, and the product evolves as it is created, the time frame for the development project is much shorter than with rigid models (Sabale and Dani 2012).

However, prototyping models are weak on analysis and design planning. While requirements are assessed as the product is developed in successive versions, there is little control over costs and resources, which can dramatically increase the financial cost of the project (Sabale and Dani 2012). Therefore, we can conclude that prototyping is ideal for larger projects and particularly for user-centric ones.

2.9 Usability Engineering Life Cycle

Usability engineering is a concept of software engineering that places usability characteristics at the center of the development process and implies that constant measurements and analysis of usability should be undertaken as the development proceeds. It is primarily related to user interface design. As a model for SDLC, it was originally proposed by Deborah Mayhew in the late 1990s.

The main objective of usability engineering is to apply structured iterative design and evaluation to all stages of the SDLC, thus ensuring constant involvement of the user within the process. The life cycle is segmented in three parts—analysis/design, development, and evaluation. Across the three sections, the successive activities are performed, in a waterfall-like process, but reflecting the existence of various iterations before the final product is released (Gabbard et al. 2003).

Crucial to this process is determining who will constitute the product's user base and what will they be doing with the product. User task analysis is the central activity at the beginning of the process and can be achieved through surveys, interviews, observation, etc. The product of such analysis consists of scenarios, potentialities, and requirements that will be taken into account for the next stages of the process. After establishing the requirements and defining them through user-centered metrics, an initial design is outlined and swiftly prototyped, so that it can be followed by extensive usability evaluation and testing (Gabbard et al. 2003).

This life cycle model is evidently user-centric, as it intends to create a system that is financially effective but also presents very high usability. The fact that it helps the development of systems that are extremely user-friendly prevents errors that derive from human misuse of the interface. Consequently, it promotes high productivity.

When considering the entirety of the life of the interface, this development life cycle, because it diminishes the need for the addition of features at a stage of the development when they have an increased cost, has the potential to decrease expenses (Gabbard et al. 2003)

2.10 The Star Life Cycle Model

The star life cycle model was proposed by Harton and Hix in the late 1980s, as the result of extensive observation of developers in real-time environments (Helms 2001). It is a particular variant of usability engineering, a user-centric set of software development guidelines, and thus rejects the rigid, step-by-step nature of waterfall-like models.

The most innovative premise is that each step in development does not necessarily fall in a fixed position within a fixed process. Instead, it is assumed that there are a number of essential stages of development, but they can be processed in various orders and various time frames, according to the specific needs of the

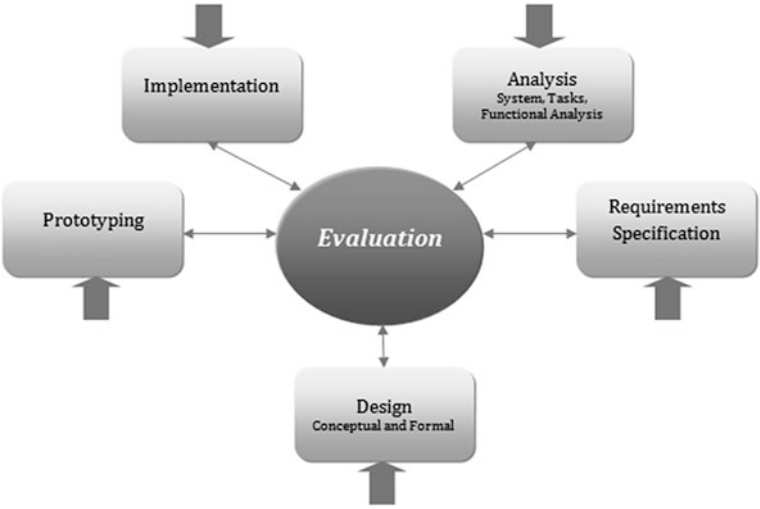

Fig. 2.8 The star life cycle model (adapted from Stone et al. 2005)

project, with the possibility of going back over a given stage numerous times or completely skipping another stage if it proves irrelevant. Thus, for example, a developer might start by experimenting with various design options and, in that process, learning more specific requirements of the project (Helms 2001).

The fundamental rule behind this premise is that each stage must be accompanied by extensive evaluation. All the stages are interrelated, and in the development process, it is possible to shift to any of them at any point, just as long as that stage is evaluated. Likewise, each action that is completed, regardless of its order, has to be thoroughly analyzed. This includes extensive testing and data collection on that particular activity, through such methods as interviewing users or observing their use of the system within the working context (Stone et al. 2005) (Fig. 2.8).

Users are positioned at the center of the development cycle and are encouraged to participate in any of the stages: at the beginning of the process, to help establish the system's requirements and define their goals and needs; during prototyping, to ensure thorough testing under working conditions; before final release, again to provide extensive testing; and after the system's delivery, to monitor any possible issues and communicate their overall experience with the system (Stone et al. 2005). This perspective clearly derives from the star model's close relationship with user interface design, as it was originally conceived within that particular context.

The model is laid out like a star, hence its name. Evaluation is at the center of the star, since it is the fundamental premise that will guide all other steps. Around the central step, we find the different possible stages of development; however, they are not connected to each other. This does not mean that there is no relationship between the different stages; what it does illustrate is that every step is interconnected through the process of evaluation (Helms 2001).

2.11 Hybrid System Development Life Cycles

An environment of increasing competitiveness demands systems that are safe and trustworthy. They also have to be adaptable and flexible to the changes that can happen at any moment, in a fast paced, ever-evolving world. This brings complications for the development process. A common response to these issues is the combination of different system development life cycles.

SDLC models each have their own peculiar characteristics, which can be both advantageous and detrimental, depending on the type of project requirements and features. Once a project is hypothesized, a model is chosen to fit its purposes. But if the particular characteristics of the project do not necessarily fit one specific model, it is possible to combine guidelines from more than one. This combination is primarily done to harness the qualities of a model and reduce its weaknesses by incorporating the strengths of another model (Rahmany 2012).

An example of combined life cycle models is the case of a development process that is being guided by the spiral model, but that later in the process demands a change in the requirements. To accommodate this need, the agile model could be incorporated (Rahmany 2012).

Madachy et al. (2006) have outlined a hybrid SDLC model which they named the scalable spiral model. The main purpose was to combine a plan-driven approach with an agile one. Development is organized into thoroughly planned increments, which take into consideration relatively stable, initial requirements. However, upon the release of each increment, it is crucial to have an agile team focusing on market, competition, and technological analysis, as well as user feedback and renegotiation of the characteristics of the next increments. The model ultimately aims at simultaneously catering for the challenges of rapid change and the need for risk management and dependability (Madachy et al. 2006).

Munassar and Govardhan (2010) have also attempted a similar approach, which they simply named the hybrid model. They picked up essential traits from such different models as waterfall, iteration, V-shaped, spiral, and extreme programming (XP). It consists of a series of seven steps that are interconnected with each other: planning, requirements (at which point risk analysis is undertaken), design, implementation (including testing), integration development, deployment, and maintenance. Although the process appears to be outlined in the same style of a waterfall approach, the relationship between the different stages is fluid and multidirectional, accounting for possible changes in requirements and the need to revise design features after testing. The authors argue that this approach would permit to combine the best characteristics of each model: It promotes good habits of define-before-design and design-before-code (like the waterfall model), while, at the same time, it avoids the dangers of rigid development by introducing early testing. It is also iterative, but still incorporates risk analysis, and the test-based approach allows for high usability (Munassar and Govardhan 2010).

2.12 Conclusion

Variety in SDLC models and frameworks is great. We have discussed the most prominent ones, but there are numerous other models, many of them hybrid in nature and designed to respond to specific needs of specific projects or simply to attempt a flawless approach by combining various models and reducing their individual weaknesses. This great variety means that the process of choosing which model to adopt for a particular development project can be complicated. Nevertheless, there are certain fundamental aspects of the project that can facilitate the decision.

The requirements of the system play a key role. If requirements are strict and immutable, the team might adopt a waterfall approach, but if requirements are expected to change often, or are not clearly defined at the start, the team will probably adopt a more agile and/or iterative approach.

The deadline for the development of the system is also an important factor. It is clear that if the schedule is tighter, a rigid step-by-step model based on extensive documentation and late testing would be unreasonably slow, thus excluding waterfall models.

The project's dimension is one of the most influential factors. The larger the project, the more rigid the model tends to be, because a large team comprised of many developers makes agile responses more complicated. The location of the teams is also a factor: If the teams involved in the project are geographically dispersed, a waterfall-like model is probably the best approach, for the clarity of its stages and tasks. An agile development, where tight communication is important, is an approach that is more beneficial to small teams working closer together.

Finally, resources should always be taken into account. Projects that involve intricate dynamics and demand the use of peculiar expertise and technology are easier to accomplish with models of strict planning, such as the waterfall (Executive Brief 2008).

Choosing the right model for a project is a crucial step of system development, so as IS continues to be fundamental to modern business and organizational contexts, SDLC models will continue to be developed, researched, and utilized.

References

Agarwal, R., Prasad, J., Tanniru, M., & Lynch, J. (2000). Risks of rapid application development. *Communications of the ACM, 43*(11), 177–188.

Ambler, S. (2009). The Agile system development life cycle (SDLC). Retrieved from http://www.ambysoft.com/essays/agileLifecycle.html

Balaji, S., Murugaiyan, M., (2012). Waterfall vs. V-Model vs. Agile: A comparative study on SDLC. *International Journal of Information Technology and Business Management 2*(1), 26–30.

Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R., Mellor, S., Schwaber,

K., Sutherland, J., & Thomas, D. (2001). Manifesto for Agile software development. Retrieved from http://agilemanifesto.org

Bhalerao, S., Puntambekar, D., & Ingle, M. (2009). Generalizing Agile software development life cycle. *International Journal on Computer Science and Engineering, 1*(3), 222–226.

Boehm, B. (1988). A Spiral model of software development and enhancement. *IEEE Computer, May 1988*, pp. 61–72.

Boehm, B. (2000). Spiral development: experience, principles and refinements (Special Report CMU/SEI-2000-SR-008). Carnegie Mellon University.

Carr, M., & Verner, J. (1997). Prototyping and software development approaches. Department of Information Systems, Hong Kong: City University of Hong Kong.

Cockburn, A. (2008). Using both incremental and iterative development. *STSC Cross Talk, 21*(5), 27–30.

Cohen, S., Dori, D., & de Uzi Haan, A. (2010). A software system development Life Cycle model for improved stakeholders' communication and collaboration. *International Journal of Computers Communications & Control, 1*, 23–44.

Stone, D., Jarrett, C., Woodroffe, M., & Minocha, S. (2005). Introducing user interface design. In D. Stone, C. Jarrett, M. Woodroffe, & S. Minocha (Eds.), *User interface design and evaluation* (pp. 3–24). San Francisco: Elsevier.

Durrani, Q. S., & Qureshi, S. A. (2012). Usability engineering practices in SDLC. *Proceedings of the 2012 International Conference on Communications and Information Technology* (*ICCT*) (pp. 319–324).

Executive Brief (2008). *Which Life Cycle is best for your project?* Retrieved from http://www.executivebrief.com

Floyd, C. (1984). A systematic look at prototyping. In R. Budde, K. Kuhlenkamp, L. Mathiassen, & H. Züllighoven (Eds.), *Approaches to Prototyping* (pp. 1–18). Berlin: Springer.

Gabbard, J., Hix, D., Swan, E., Livingston, M., Hollerer, T., Julier, S., Brown, D., & Baillot, Y. (2003). Usability engineering for complex interactive systems development. *Proceedings of Human Systems Integration Symposium 2003, Engineering for Usability* (pp. 1–13).

Gottesdiener, E. (1995). RAD realities: beyond the hype to how RAD really works. *Application Development Trends, August 1995* (pp. 28–38).

Helms, J. (2001). Developing and evaluating the (LUCID/Star)*Usability Engineering Process Model (Master's thesis). Retrieved from http://scholar.lib.vt.edu/theses/available/etd-05102001-190814/unrestricted/jhelmsthesisnew.pdf

Jirava, P. (2004). System development life cycle. In *Scientific Papers of the University of Pardubice Series D.* (pp. 118–125). Pardubice: Univerzita Pardubice.

Madachy, R., Boehm, B., & Lane, J. (2006). Spiral lifecycle increment modeling for new hybrid processes. In Q. Wang, D. Pfahl, D. Raffo, & P. Wernick (Eds.), *Software process change* (pp. 167–177). Berlin: Springer.

Massey, V., & Satao, K. (2012). Comparing various SDLC models and the new proposed model on the basis of available methodology. *International Journal of Advanced Research in Computer Science and Software Engineering, 2*(4), 170–177.

Mathur, S., & Malik, S. (2010). Advancements in the V-Model. *International Journal of Computer Applications IJCA, 1*(12), 30–35.

Munassar, N., & Govardhan, A. (2010a). Comparison between five models of software engineering. *International Journal of Computer Science Issues, 7*(5), 94–101.

Munassar, N., & Govardhan, A. (2010b). Hybrid model for software development processes. *Proceedings of the 11th International Arab Conference on Information Technology*.

Rahmany, N. (2012). The differences between life cycle models—Advantages and disadvantages. Retrieved from http://narbit.wordpress.com/2012/06/10/the-differences-between-life-cycle-models-advantages-and-disadvantages/

Royce, W. (1970). Managing the development of large software systems. *Proceedings of IEEE WESCON* (pp. 1–9).

Sabale, R., & Dani, A. (2012). Comparative study of prototype model for software engineering with system development Life Cycle. *IOSR Journal of Engineering, 2*(7), 21–24.

Skidmore, S. (2006). The V-Model. *Student Accountant, January 2006* (pp. 48–49).
Tetlay, A., & John, P. (2009). Determining the lines of system maturity, system readiness and capability readiness in the system development lifecycle. Presented at the 7th Annual Conference on Systems Engineering Research (CSER09), Loughborough University.
Texas Project Delivery Framework, U.S. Department of Information Resources (2008). System development Life Cycle guide. Texas Department of Information Resources. Retrieved from http://www.dir.texas.gov/SiteCollectionDocuments/IT%20Leadership/Framework/Framework%20Extensions/SDLC/SDLC_guide.pdf
Tilloo, R. (2013). What is incremental model in software engineering? Retrieved from http://www.technotrice.com/incremental-model-in-software-engineering

http://www.springer.com/978-1-4614-9253-5

High Level Models and Methodologies for Information Systems

Isaias, P.; Issa, T.
2015, XVI, 145 p. 48 illus., Hardcover
ISBN: 978-1-4614-9253-5