Computing Handbook
Information Systems and Information Technology
Heikki Topi, Allen Tucker

From Waterfall to Agile: A Review of Approaches to Systems Analysis and Design

Publication details
Joey F. George
Published online on: 14 May 2014

Accessed on: 12 Nov 2018

PLEASE SCROLL DOWN FOR DOCUMENT

Full terms and conditions of use: https://www.routledgehandbooks.com/legal-notices/terms

This Document PDF may be used for research, teaching and private study purposes. Any substantial or systematic reproductions, re-distribution, re-selling, loan or sub-licensing, systematic supply or distribution in any form to anyone is expressly forbidden.

The publisher does not give any warranty express or implied or make any representation that the contents will be complete or accurate or up to date. The publisher shall not be liable for an loss, actions, claims, proceedings, demand or costs or damages whatsoever or howsoever caused arising directly or indirectly in connection with or arising out of the use of this material.
28

From Waterfall to Agile:
A Review of Approaches
to Systems Analysis and Design

28.1 Introduction
While there will always be some debate about when and where the first information system for business was developed, it is generally agreed that the first such system in the United Kingdom was developed at J. Lyons & Sons. In the United States, the first administrative information system was General Electric’s payroll system, developed in 1954. Now, 60 years later, the people who designed and built those systems would scarcely recognize the approaches and tools currently used to develop information systems. Systems development, which will be used in this chapter synonymously with systems analysis and design, has changed quite a bit in the past six decades.

It is the premise of this chapter that systems development has evolved through three distinct phases (Figure 28.1). Initially, development was a craft, where each system was unique and not replicable. Changes in technology and in attitude, and concerns from organizational management, led to the second phase, where development was seen as a feat of engineering. Planning, structure, and documentation governed the process. But, according to critics, the approach grew too big, too rigid, and too slow. New approaches began to emerge, including the rational unified process (RUP) and participatory design. Eventually, a new phase dawned, and the focus shifted from plan-based to small, fast, and agile. Today, the plan-based and agile approaches coexist side-by-side. This coexistence is not tension free, however, and this uneasy state of affairs will no doubt be followed by another phase, although what
that will be is not so clear. In this chapter, we will present an overview of the three phases of systems development approaches, and we will close with some conjecture on what the next step might be. Each phase will be presented with reference to specific methodologies and tools that emerged during that phase, but in each case, the list is only representative and not comprehensive. A complete list of all of the development methodologies that have been created and used in the past 60 years is far beyond the scope of this chapter.

### 28.2 Systems Development as Craft

In the early years, especially in the time before early third-generation languages such as COBOL, software development more closely resembled an art form than a production business process. Early systems development often occurred in a chaotic and haphazard manner, depending on the skills and experience of the people performing the work (Green and DiCaterino, 1988). In this era, developers were technically trained to work on particular hardware systems, which were incompatible with systems manufactured by other vendors. Many capitalized on peculiarities of a particular machine, using their proprietary machine and assembly languages, for writing and optimizing code. Developers had little understanding of the business, and user needs were poorly understood, if considered at all. There were no development methodologies. “The approach programmers took to development was typically individualistic, often resulting in poor control and management of projects” (Avison and Fitzgerald, 2003, p. 79). Schedules, budgets, and functionality were all unpredictable, and maintenance was a nightmare, as documentation was largely non-existent. When the system designer left the firm, few of those remaining had clear ideas about how to maintain the systems left behind. As technology improved, with the development of procedural programming languages, database management systems, and as management grew tired of the chaos and waste, this era started to come to an end. Calls for structured programming (Dijkstra, 1968) were accompanied by calls for a more structured and predictable approach to systems development.

### 28.3 Systems Development as Engineering

The structured approach emerged in the 1970s and continued to dominate systems development well into the 1990s. At the heart of the structured approach was a focus on designing and building systems the same way other artifacts, such as bridges and buildings, are designed and built. Like these artifacts, systems would start as concepts, which would then be modeled using several different tools, and these logical models would be converted to physical plans that would guide the construction of the system. The overall process would become an engineering process, and systems developers would become software engineers. Development would become systematic, predictable, repetitive and repeatable, measurable, heavily documented, and reliable. As a result, the productivity of programmers and developers would increase, as would the quality of the resulting systems.
Although many have been associated with it, maybe the best known advocate for the structured approach was Edward Yourdon. Yourdon (1989, p. 3) listed the major components of the structured approach as:

**Structured analysis**: A collection of graphical modeling tools that allows the systems analyst to replace the classical functional specification (text based) with a model that the users can actually understand.

**Top-down design and implementation**: The strategy of designing a system by breaking it into major functions, and then breaking those into smaller subfunctions, and so on, until the eventual implementation can be expressed in terms of program statements.

**Structured design**: A set of guidelines and techniques to help the designer distinguish between “good” and “bad” designs. Proper use of structured design will lead to a system composed of small, highly independent modules, each of which is responsible for carrying out one small single-purpose function.

**Structured programming**: An approach to programming that constructs all program logic from combinations of three basic forms (sequence, loops, and conditional statements).

The graphical models that structured analysis relied on included data flow diagrams and entity relationship diagrams. Structured design relied on tools such as structure charts and on concepts such as modularization, coupling, and cohesion. Structured programming made use of flow charts and decision tables. Despite the intuitive appeal of bringing order to chaos that structured analysis and design promised, adoption by organizational software development groups was generally slow (Fichman and Kemerer, 1993). For many, it was difficult to accept discipline where it had been missing before. The learning curve for mastering the many diagrams and other tools that supported structured analysis and design was substantial.

### 28.3.1 Systems Development Life Cycle

At the core of structured analysis and design was the systems development life cycle (SDLC). The SDLC embodies the key ideas of any product life cycle. At some point, a need is recognized; the need is investigated and a solution is designed, built, and released. Eventually, the solution reaches its maximum usefulness and begins to decline. Eventually, it must be replaced and a new solution must be created. One cycle has ended and a new one begins.

Although no two people in the IS field, either academics or practitioners, seem to agree about the exact form and contents of the SDLC, it does constitute a useful model of the systems development process. Figure 28.2 shows a generic SDLC model, typically referred to as a waterfall model.
Although different versions of the waterfall SDLC have different numbers of phases, the generic model here has five phases that capture the basic structure of the systems development process. The first phase, planning, represents all of the activities that result in the identification of a need for a system. Once the need is recognized, it is important to gain a clear understanding of the problem or opportunity that sparked the need. The existing system should be analyzed, and any additional requirements should be determined in this phase, analysis. Useful tools for analysis include joint application design (JAD) and prototyping (explained in more detail later in this chapter). Once the existing situation is understood, it can be compared to where things should or ought to be. At that point, a solution can be designed, and this is what occurs in the design phase. Design involves both high-level logical design and physical design tied to a particular computing platform and operating environment. Once designed, the solution can be constructed, tested, and installed or released, the primary activities of implementation. Once the solution has been installed in an organization, the maintenance phase begins. In maintenance, the work centers on keeping the solution viable, including making any changes required due to the changing business environment and changing regulations and legal conditions. In maintenance, programmers also work to correct errors in the system and to optimize system performance. At some point, the solution is no longer viable and a new solution must be sought, ending one cycle and beginning another.

The end of each phase is a major development project milestone. As a phase ends, a decision must be made to stop or to continue. If the decision is made to continue, the next phase can begin. Notice that there are two sets of arrows connecting the phases in Figure 28.2. One set goes downhill, connecting the phases in the order in which they were described previously. Yet, the systems development process is rarely this cleanly structured. Early waterfall models contained only the downhill set of arrows, and this was the source of much early criticism of the model. In practice, it is much more typical for developers to repeat a phase, or to return to an earlier phase, as system requirements change and as work completed earlier needs to be reexamined or even redone. Iterations within the life cycle are very common. The second set of arrows, the ones going backward and uphill, has been included in Figure 28.2 to represent the possibility and likelihood of iterations within the life cycle. The flow of work is not linear, as would be the case in a true waterfall, but instead it is cyclical. The uphill arrows allow the work flow to move from one phase to an earlier phase, and in conjunction with the downhill arrows, allow for numerous iterations between phases.

The name waterfall refers to the shape of the model, in that the work flow cascades from one phase down to the next until the end is reached. As indicated previously, there are many variations of the standard waterfall SDLC. Techniques such as rapid application development (RAD) (Martin, 1991) used modified versions of the waterfall SDLC. RAD will be discussed in a later section.

Another way to visualize the SDLC is the spiral model (see Boehm, 1988), shown in Figure 28.3. The spiral model is also referred to as an evolutionary model. The spiral model combines the iterative nature of development with the systematic, stepwise approach of the waterfall model. Software is seen as being developed in a series of incremental releases. Early releases might be prototypes, but increasingly, the versions of the software developed are more complete. As more complete versions of the software are released, what was conceptualized as maintenance in the waterfall model is conceptualized here as subsequent passes around the spiral.

According to the model in Figure 28.3, each pass around the spiral, whether for initial iterative development or for later maintenance, moves through six different tasks (Pressman, 1997). The first is customer communication, which involves establishing effective communication between the developer and the customer or user. The second task is planning, where resources, time-lines, and other project information is defined. The third task is risk assessment. Here, technical and management risks are assessed, and if the risks are too high, the project can be ended at the go-no-go axis. The fourth task is engineering, which represents the building of the software application. Engineering corresponds to design in the waterfall model. The fifth task is construction and release, which contains many of the same activities as were included in the implementation phase earlier: construction, testing, installation, and providing
From Waterfall to Agile

user support. The sixth task is customer evaluation. The evaluation process elicits feedback on the engineering and construction and release tasks.

While the waterfall model is a simple and easy to understand representation of the systems development process, the spiral model presents a more complex and hence more realistic view. The spiral model adds iteration, risk assessment, and a more sophisticated view of maintenance to the basic waterfall model. Many of the basic tasks central to system development are the same in both models, however, as would be expected. The spiral model has also had great success in practice. Boehm (1988) cites adherence to the model as the cause of productivity improvements as high as 50% in development projects at TRW (an aerospace and automotive company, purchased by Northrop Grumman in 2002), where Boehm’s particular version of the model was developed and used. The spiral model is not suited to all software development efforts, however. It best fits large-scale development projects. Another limitation in practice is its demands for considerable risk assessment expertise in order to succeed, an issue of particular concern to both Boehm and Pressman.

28.3.2 Structured Tools and Techniques

As the structured approach to systems development became more widely adopted, various tools and techniques were developed to enrich the approach and to make it easier to follow. People began to automate the graphical tools that supported the approach, and eventually the automated tools were combined with a rigorous methodology in computer-assisted software engineering, or CASE, tools. Others borrowed techniques from engineering, such as prototyping, where a smaller working model of an artifact was built as a way to help system users articulate the system’s requirements. Other techniques created to assist the requirements determination process included JAD, a method that brought together the key stakeholders (or their representatives) in one place at one time, to help clarify what was desired in the new or revamped system. Each of these tools and techniques, all of which strongly supported the structured approach to systems development, will be discussed in more detail in the paragraphs that follow.

CASE tools were developed as a way to use the power of information technology to support the software development process. CASE tools are bundles of software tools that make analysis, design, documentation, and coding consistent, complete, and easier to accomplish. CASE packages built for the structured approach included tools for creating and maintaining data flow diagrams, entity-relationship diagrams, and structure charts, among others. The diagramming tools were linked in that each object created in one of the tools was entered into the central repository around which the CASE product was built. The repository stored data about all of the different elements that make up the system analysis.

FIGURE 28.3 A spiral systems development life cycle.
and design. CASE tools typically also contained facilities that allowed for screen and report design, as well as facilities that generated database structures and program code. Many structured CASE tools also allowed checking for consistency, completeness, and design errors.

Although structured CASE tools were expensive, costing several thousand dollars per copy, including training and support, they were generally believed to increase developer productivity. A case study of the implementation of Texas Instruments’ information engineering facility (IEF) at a British firm in 1989–1990 reported productivity improvements of 85% and system delivery rate increases of around 200% (Finlay and Mitchell, 1994). Developers who used structured CASE were more likely to follow a structured systems development methodology than their counterparts who did not use CASE (Lending and Chervany, 1998). However, given the high cost and steep learning curve, relatively few firms adopted structured CASE tools, and for those that did, relatively few of the developers used CASE (Iivari, 1996; Lending and Chervany, 1998). Where CASE tools were adopted, there was strong management support for the approach, the tools were perceived as having relative advantage over non-CASE methods, and use tended to be mandated, not voluntary (Iivari, 1996).

Prototyping was a process borrowed from engineering, in which a scale model of an object or system was developed, both as a test-of-concept and as a device to facilitate communication between designer and client (Gavurin, 1991; Naumann and Jenkins, 1982). The software tools used for prototyping could be as simple as paint programs or as complex as CASE tools that allowed users to enter data and navigate between screens. The main idea behind prototyping as a technique was the provision of a means by which developers could transform user requirements into objects that users could see, touch, and use. Users could then provide feedback to developers about the prototype, about what worked and what did not, and about what else they would have liked to see. The developer would then take that feedback and use it to modify the prototype, beginning another round of iteration between user and developer. At some point, it was decided to either keep the prototype as the basis for the production system (evolutionary prototyping) or to throw it away (sometimes called mock-up design prototyping), while using the knowledge gained in the prototyping process to construct the production system. Prototyping became very popular in system development efforts, with over 70% of firms surveyed reporting the adoption of prototyping by 1995 (Hardgrave, 1995). Studies of prototyping revealed that, compared to an SDLC approach that did not use it, prototyping was associated with higher levels of both user and developer satisfaction, higher levels of user commitment, improved communication with users, earlier detection of errors, better code, and improvements in the development process in terms of shorter development time and less effort for developers (Beyton-Davies et al., 1999; Cerpa and Verner, 1996; Mahmood, 1987). By its very nature, prototyping encouraged meaningful and frequent interaction between user and developer. Prototyping was not without problems, however. Prototyping was associated with difficulties in managing user expectations, enticing users to commit to fully participate in the prototyping effort, and with project management generally (Beyton-Davies et al., 1999).

JAD was a technique developed primarily to address the issues involved in determining user requirements for a system (Wood and Silver, 1995). Traditionally, a developer determined requirements by interviewing users, studying existing systems, perusing input form, reports, and other documents, and observing users while working. User interviews were an important source of information for what the system being developed should do and look like. Interviews, however, were difficult to schedule, as users had their own regular work to complete while assisting the developer. Contradictions between users also had to be reconciled, which required follow-up interviews. The process could be very time consuming and frustrating.

JAD was developed to deal with the difficulties in scheduling interviews and reconciling the information collected from them. In JAD, key users met with management representatives and systems developers, sometimes for a day, sometimes for an entire week, to determine requirements for a system. The process was very structured, typically led by a trained JAD facilitator whose primary role it was to keep the process moving. JAD essentially allowed for many interviews to be conducted at once and for contradictions to be reconciled on the spot (although not always possible). A JAD session was typically held off-site to minimize the distractions participants would experience if working at the office. There has
been little empirical research into the effectiveness of JAD. One study reported that JAD was superior to traditional methods, with respect to the quality of user–designer interactions, effectiveness of consensus management, and user acceptance of design specifications (Purvis and Sambamurthy, 1997). Other studies combined JAD with other approaches for managing group work, from electronic group support to the nominal group technique. JADs run with electronic group support had more equal participation among group members but lacked the session discipline and conflict resolution common to traditional JAD (Carmel et al., 1995). JAD combined with the nominal group technique was found to be more effective than traditional JAD, although just as efficient, with higher levels of user satisfaction and better system requirements determination (Duggan and Thachenkary, 2004).

### 28.3.3 Other Structured Approaches

**Information engineering** (IE) is a structured approach to systems analysis and design originally developed by Clive Finkelstein (http://www.ies.aust.com/cbfindex.htm). He later worked with James Martin, who developed his own version of the methodology in the mid-1980s (Martin, 1986). As with many other structured approaches to systems analysis and design, IE depended on sets of related and interconnected diagramming models. These models included entity–relationship diagrams, data flow diagrams, decision trees (called action diagrams), and report and screen layout tools. IE also stressed the importance to the development process of automated tools, which allowed high levels of user participation, as well as high levels of horizontal integration (between systems) and of vertical integration (linking systems to top management goals and corporate strategy). IE had four stages: (1) information strategy planning; (2) business area analysis; (3) system design; and (4) construction. The planning phase focused on the entire enterprise, while a different business area analysis would be done for each different business area. These two processes together took from 9 to 18 months to complete. The latter stage focused on identifying the processes and data needed for each business area. Design was conducted via automated development tools, and construction was aided by code generators and other tools. At all stages of the IE process, knowledge was stored in an encyclopedia, or central repository.

Martin later worked with Texas Instruments on instantiating the methodology into a CASE tool called IEF, for information engineering facility. IEF was released in 1987. In 1997, Texas Instruments sold IEF to Sterling Software, which was later acquired by Computer Associates.

**RAD** was a development methodology also credited to James Martin (1991). The idea behind RAD was captured in its name: dramatically shorten the time necessary for systems to be developed. Complex business systems typically took years to develop using structured analysis and design processes, but with the dynamic pace of business in an expanding global economy, firms could not wait for systems that, once completed, might no longer be adequate models of their business processes they modeled. According to Martin, a system that would take 2 years to complete following the traditional approach might instead be completed in 6 months following the RAD approach.

RAD was purported to save time and other related resources, like money, because of how it was designed to work. First RAD made heavy use of JAD and of CASE tools to support prototyping. The first JAD meetings in a RAD effort might involve prototyping very early in the process, before a more traditional JAD would. The prototype developed also tended to become the basis for the production system, rather than being thrown away, as discussed previously. Second, the prototyping process in a RAD required more intensive participation from users in a true partnership with developers. Users might also have become involved in the design process itself instead of ending their participation after requirements determination was complete, which was more typically the case using traditional approaches. Martin (1991) also argued that developers had to be trained in the right skills and methodologies, and management had to lend its complete support, for RAD to be successful.

As good as it might sound, RAD was not without its problems. Sometimes, due to the speed with which systems were developed, some of the basics of traditional systems development were overlooked. These included interface consistency across the system, programming standards such as documentation.
and data naming standards, upward scalability to serve larger numbers of more diverse users, and planning for system administration chores such as database maintenance and organization, backup and recovery, and so on (Bourne, 1994).

### 28.3.4 Alternative Development Approaches and Practices

Even though it was not universally adopted, the structured approach to systems development dominated development work for many years. Before the transition to agile methods that occurred in the late 1990s and the early part of the twenty-first century, there existed several alternatives to the structured approach. One, the RUP, was created to guide object-oriented systems development efforts. Another, participatory design, focused more on people and creative hands-on techniques for development rather than focusing on the technology and using it to support development. A third approach, reuse, is really not so much a separate methodology as it is a practice created to make any development process more efficient through reusing existing software code and components. Each of these approaches and practices are discussed in more detail in the sections that follow.

**RUP** was a development methodology that emerged from the object-oriented perspective on analysis and design. It had its origins in the work of Jacobson et al. (1999), although the process now is owned by IBM (http://www-01.ibm.com/software/awdtools/rup/). The approach was built around iteration, with a focus on being risk-driven (Aked, 2003; Ambler, 2005). The risk-driven approach refers to the practice of tackling head on concerns that could adversely affect the development project, rather than dealing with easy issues first and dealing with risk later (Aked, 2003). The methodology crossed four phases (inception, elaboration, construction, and transition) with nine disciplines, which described the different types of work to be done. These nine disciplines are: (1) business modeling; (2) requirements; (3) analysis and design; (4) implementation; (5) test; (6) deployment; (7) configuration and change management; (8) project management; and (9) environment. The phases and disciplines are often shown in what is called a “hump diagram,” where the relative effort devoted to each discipline within each phase is shown (Figure 28.4 shows only the first six disciplines). Each phase was also divided into a number of iterations. At the end of each phase, there was a go/no go decision, much as was the case in an SDLC from the structured approach.

The inception phase included defining scope, determining the feasibility of the project, understanding user requirements, and preparing a software development plan. In the elaboration phase, detailed

![Iterative development](http://en.wikipedia.org/wiki/File:Development-iterative.gif)
user requirements and a baseline systems architecture were developed. Analysis and design activities constituted the bulk of the elaboration phase. In the construction phase, the software was coded, tested, and documented. In the transition phase, the system was deployed and the users were trained and supported. The construction phase was generally the longest and the most resource intensive. The elaboration phase was also long but less resource intensive, relatively speaking. The transition phase was resource intensive but short. The inception phase was short and the least resource intensive.

The nine disciplines were one way to package the skills and processes needed for successful systems development. The first six, from business modeling to deployment, corresponded to a typical development process, from planning through analysis, design, testing, and implementation. Configuration and change management corresponded to the ongoing maintenance process, which follows implementation. Both project management and environment ran more or less throughout the development process. While the former is self-explanatory, the latter refers to making sure the appropriate guidance and tools are available to the development (and maintenance) team throughout the process (Ambler, 2005). As RUP came out of the object-oriented perspective, for implementation it relied heavily on the unified modeling language (UML) (Booch et al., 2005). RUP and UML were captured in the Rational Rose CASE tool, which is now sold by IBM as the Rational series of tools. Stories of RUP use in organizations and the resulting successes and problems were typically told in case study form (see Hanssen et al., 2005, and Lee, 2006, for examples).

Participatory design was a very different approach to systems development than anything presented so far. Developed in Northern Europe, participatory design focused primarily on the system’s users and how the system would affect their work lives. In some cases, the entire user community might play an active role in the systems development process, while in others, users might select representatives to represent them. Typically, system developers worked for the users. Management and outside consultants provided advice but did not control the process. Participatory design efforts included a repertoire of flexible practices and general guidelines. Running through the repertoire were the twin themes of mutual reciprocal learning, in which users and developers teach each other about work and technology, respectively, and design by doing, which resembles prototyping but does not necessarily require sophisticated computing to implement (Carmel et al., 1993). While well known in Northern Europe, especially in the Nordic countries, participatory design was rarely followed elsewhere.

Reuse was the use of previously written software resources in new applications. As so many bits and pieces of applications are relatively generic across applications, it seems intuitive that great savings could be achieved in many areas if those generic bits and pieces did not have to be written anew each time they were needed. Reuse should increase programmer productivity, as being able to use existing software for some functions meant they could perform more work in the same amount of time. Reuse should also decrease development time, minimizing schedule overruns. Because existing pieces of software have already been tested, reusing them should also result in higher quality software with lower defect rates, which is easier to maintain.

Although reuse could conceivably apply to many different aspects of systems, typically it was most commonly applied to object-oriented code and components. In object-oriented code, both data and function are combined in one item. For example, an employee object would contain both the data about employees and the instructions necessary for calculating payroll for a variety of job types. The object could be used in any application that dealt with employees, but if changes had to be made in calculating payroll for different type employees, the changes would only have to be made to the object and not to the various applications that used it. By definition, using the employee object in more than one application constitutes reuse. As with CASE tools and structured development, developers were slow to adopt object-orientation (Fichman and Kemerer, 1993), although that has changed dramatically in the past few years. Today, four of the five most used programming languages are object-oriented (Java, C#, C++, and Objective-C, with the other being C) (TIOBE, 2012). Components are general-purpose pieces of software that can be used interchangeably in many different programs. Components can be as small as objects or as large as pieces of software that handle single business functions, such as currency conversion. The idea behind component-based development is the assembly of an application.
from many different components at many different levels of complexity and size. Reuse of objects and components has proven to be effective, resulting in increased productivity, reduced defect density, and reduced rework (Basili et al., 1996). Reuse has its problems, however. In a study of 24 European companies that undertook reuse projects, one-third of the projects failed (Maurizio et al., 2002). The reasons for failure were not technical—they were managerial. How changes to work processes that govern reuse are implemented and managed are key to its success (Sherif et al., 2006).

28.4 Agile Development

Although the engineering approach remains in use today, to many critics, it has become bloated and slow, no longer as useful in a global economy that runs on Internet time. The convergence of the object-oriented approach and the Internet economy set the stage for the current major phase in systems development. Relying on object-orientation and the need for speed, the new approach sacrifices the milestones and multiple phases of the engineering approach in favor of close cooperation between developers and clients, combining many life cycle phases into few phases, and multiple rapid releases of software. Although there are many individual methods that reflect the new approach, collectively they are called the agile methodologies.

Many different individual methodologies come under the umbrella of agile methodologies. In February 2001, many of the proponents of these alternative approaches to systems analysis and design met in Utah, USA, to reach a consensus on many of the underlying principles their various approaches contained. This meeting led to a document they called “The Agile Manifesto” (Beck et al., 2001). The agile methodologies share 12 principles (http://www.agilemanifesto.org/principles.html), which can be summarized in three key points: (1) a focus on adaptive rather than predictive methodologies, (2) a focus on people rather than roles, and (3) a focus on self-adaptive processes (Valacich et al., 2012). Agile adherents argue that software development methodologies adapted from engineering generally are not a good fit for the reality of developing software. In the engineering disciplines, such as civil engineering, requirements tend to be well understood. Once the creative and difficult work of design is completed, construction becomes very predictable (Fowler, 2005). In addition, construction may account for as much as 90% of the total project effort. For software, on the other hand, requirements are rarely understood well, and they change continually during the lifetime of the project. Construction may account for as little as 15% of the total project effort, leaving design to constitute as much as 50% of the project effort (McConnell, 1996), depending on the project type. Applying techniques that work well for predictable, stable projects, like building a bridge, tend not to work well for fluid, design-heavy projects like writing software, say the agile methodology proponents. One mechanism for dealing with a lack of predictability, which all agile methodologies share, is iterative development. Iterative development focuses on the frequent production of working versions of a system that have a subset of the total number of required features. Iterative development provides feedback to customers and developers alike.

Second, the focus on people in agile methodologies is a focus on individuals rather than on the roles that people perform. The roles that people fill, of systems analyst or tester or manager, are not as important as the individuals who fill those roles. Fowler (2005) argues that the focus on engineering principles applied to systems development has resulted in a view of people as interchangeable units instead of a view of people as talented individuals, each of which has something unique to bring to the development team. Third, the agile methodologies also promote a self-adaptive software development process. As software is developed, the process used to develop it should be refined and improved. Development teams can do this through a review process, often associated with the completion of iterations.

28.4.1 eXtreme Programming

One of the best known and most written about agile methodologies is called eXtreme programming. Developed by Kent Beck in the late 1990s (Beck, 2000), eXtreme programming illustrates many of the central philosophies of this new approach to systems development. eXtreme programming is
distinguished by its short development cycles, its incremental planning approach, its focus on automated tests written by programmers and customers to monitor the process of development, and its reliance on an evolutionary approach to development that lasts throughout the lifetime of the system. One of the key emphases of eXtreme programming is having a customer on-site during the development process. Another is its use of two-person programming teams.

Under this approach, users generate stories that serve as the basis for system functions and features. These user requirements are not consolidated into a single, detailed and complete design specification, as would be expected in an engineering-based development effort. Instead, requirements serve as the basis for a design, which is captured in code and then tested. Coding and testing are intimately related parts of the same process. The programmers who write the code also write the tests. The emphasis is on testing those things that can break or go wrong, not on testing everything. The overall philosophy behind eXtreme programming is that code will be integrated into the system it is being developed for and tested within a few hours after it has been written. If all the tests run successfully, then development proceeds. If not, the code is reworked until the tests are successful. Pair programming makes the code-and-test process work smoothly. All coding and testing is done by two people working together, writing code and writing tests. The programmers work together on the problem they are trying to solve, exchanging information and insight and sharing skills.

To continually improve the quality of the design, eXtreme programming relies on a practice called simple design. Simple design is what it sounds like: keeping the design simple. According to Beck (2000), it is common for developers and programmers to look ahead and to try to anticipate changes in how the system will work and to design accordingly. Many times, those anticipated future conditions never materialize, and the time and effort that went into designing for the uncertain future is wasted. Under the eXtreme programming approach, design should focus on solving the immediate problem, not on solving future problems that may or may not occur.

28.4.2 Scrum

Scrum originated in 1995 and was developed by Jeff Sutherland and Ken Schwaber (Schwaber and Sutherland, 2011). Scrum represents a framework that includes Scrum teams and their associated roles, events, artifacts, and rules. Each team consists of three roles: the product owner, the development team, and the Scrum master. The owner is essentially accountable for the product and the work that produces it. The development team is small, within the preferred range of 3–9. The Scrum master is there to teach and enforce the rules (which vary by team and by product). Scrum is designed for speed and for multiple functional product releases. The primary unit is the sprint, a 1 month (or less) unit of effort. Each sprint is a complete project in and of itself. It starts with an 8 hour sprint planning meeting, which focuses on two questions: What will need to be delivered by the end of the sprint and how will the work needed to do that be accomplished? The sprint goal provides guidance for the team for the duration of the sprint. During the sprint, there is a daily Scrum, a 15 min meeting held to essentially evaluate what progress has been made within the past 24 hours and what still needs to be done. At the end of the sprint, there are two other meetings, the sprint review (4 hours) and the sprint retrospective (3 hours). While the review focuses on the product, what has been accomplished, and what needs to be done in the next sprint, the retrospective is more broad. It focuses as well on the performance of the team and how it can improve in the next sprint. There are three primary artifacts in the Scrum process. The first is the product backlog. This is an ordered list of everything that might be included in the product, that is, a list of potential requirements. The list includes “all features, functions, requirements, enhancements and fixes (Schwaber and Sutherland, 2011, p. 12)” that make up all the changes to be made to the product. The sprint backlog is a subset of the product backlog, consisting of only those items to be addressed in a particular sprint. Finally, the increment is the sum of all the product backlog items completed during a sprint. Each increment must be in complete enough form to be useable, whether or not the product owner decides to release it. It is called an increment because it represents an increment
of total functionality for the product. Each increment is thoroughly tested, not only as a standalone but also in conjunction with all prior increments.

28.4.3 Agile Performance

Several studies have investigated agile methods in practice. One study, a survey of over 100 agile projects, found three primary critical success factors for agile development (Chow and Cao, 2008). The first is delivery strategy, which refers to the continuous delivery of working software in short time scales. The second is following agile software engineering practices. That means managers and programmers must continually focus on technical excellence and simple design. The third critical success factor is team capability, which refers to the agile principle of building projects around motivated individuals.

Another study found that, once implemented, agile methods can lead to improved job satisfaction and productivity on the part of programmers (Dyba and Dingsoyr, 2008). They can also lead to increased customer satisfaction, even though the role of on-site customer representative can be tiring and so not sustainable for very long. Agile methods tend to work best in small projects. In some instances, it may make sense to combine them with traditional development methods.

A detailed study of one agile development effort showed that some of the key principles of agile development had to be modified to help ensure success (Fruhling and DeVreede, 2006). For example, in the agile project studied, pair programming was not always used, especially when resources were needed elsewhere. Second, the process of writing the test case first and then the code was followed until the system became too complex. Third, the customer was not located in the same place as the programmers. Instead, the customer stayed in contact through regular meetings and continual e-mail and phone communication. Even with these modifications, the resulting system was considered a success—fewer than 10 updates were issued due to errors and none were issued due to implementing the wrong functionalities. Working together, the users and developers were able to clarify system requirements and create a user interface that was easy to learn and use.

28.5 Reconciliation and the Next Phase

Although the agile approach to systems development has many adherents who believe it is the only viable approach, others have found that agile works best for certain types of development projects. Traditional, or plan-based, approaches work best for a different set of development projects. For example, Boehm and Turner (2004) argue that agile methods are better for projects that involve small teams and small products; that agile methods are untested on safety-critical products; that the approach works best in a dynamic as opposed to stable environment; that agile projects require a critical mass of scarce agility experts; and that the approach works best where the participants do well in a work environment that is openly defined and even chaotic. Traditional approaches, then, work best for large, complex, and critical products where the product environment is stable and relatively few experts are needed during the course of development and where people are most comfortable in a work environment where their roles are well-defined and practices and procedures are clear. Additionally, agile methods are thought to work best where participants are collocated, while plan-based methods can function just as well where participants are physically distributed. Meanwhile, there is some evidence that agile methods may be used successfully in environments and for products that do not fit the constraints outlined earlier (Abrahamsson et al., 2009). Similarly, some of the practices thought central to agility, such as having an on-site customer and pair programming, may not contribute to agility as much as previously thought (Conboy, 2009).

In fact, there seems to be a movement toward using mixed approaches for systems development, where some aspects of plan-based and agile approaches are used together. Port and Bui (2009) showed through simulation that hybrid approaches yielded better results than either traditional or agile approaches alone. Results were measured by the value and the cost of the requirements implemented during the
simulation from both pure and hybrid approaches. In 2008, Baskerville et al. studied three software development companies where parts of the development process were supported through Scrum, while other parts were carried out using traditional plan-based methods, including a heavy emphasis on project management techniques (Baskerville et al., 2009). The people they interviewed stressed the need for alignment between the two approaches, where alignment was “described as necessary to ensure that the work carried out by the Scrum teams [was] in line with the overall scope document, budget, and project plan (Baskerville et al., 2009, p. 551).” The Scrum process and teams existed within a larger wrapper consisting of the traditional project management structure. Baskerville et al. (2009) speculated that the next stage in the overall march of systems development processes, which they called “post-agility,” will blend the goals of agility and the plan-based approach through new ways of organizing, managing, and communicating, and through new methods and software tools. Agility would no longer be a separate approach but would be deeply incorporated into all systems development methodologies. As they pointed out, the post-agility era would also be the post-plan-driven era.

28.6 Future Research Opportunities

The study of information systems development—of the methodologies, methods, and tools that have been created over the years to facilitate and support the process—was once the primary topic of research in the information systems field (Sidorova et al., 2008). Today, the topic is less studied, but it remains an important part of the field. There are still several interesting questions related to systems development that need to be investigated. This chapter suggests at least three such areas: (1) comparisons of different approaches to systems development, (2) investigations into the practices and processes currently used in industry, and (3) speculative studies of what will constitute the next stage in the evolution of system development approaches.

In the Management Information Systems (MIS) field, there has been a long tradition of studies that compare two or more approaches to systems development (see, for example, Howard et al., 1999). These studies have typically been experimental in nature, where different groups or individuals have used competing methods or tools to address a specific systems development problem. While many of these studies have compared different aspects of the structured plan-driven approach, relatively few have compared the structured approach to the object-oriented approach. Fewer still have compared plan-driven and agile approaches. This is the case even though agile methods have existed for over a decade, as of this writing. There is a clear need for systematic experimental studies that compare different approaches to systems development to each other. For both research and practical reasons, we should strive to understand the relative strengths and weaknesses of the various approaches.

A second area of research that has great potential is the study of how organizations actually approach systems development. While agile methods have received much attention in the trade press, to what extent have organizations actually adopted them? We know from past research that industry is slow to adopt new approaches to development (Fichman and Kemerer, 1993). Is that also the case for agile methods today? To what extent do organizations adopt agile methods for some projects but plan-driven approaches for others? And on what basis do they choose which approach to use? With the widespread practice of organizations outsourcing many aspects of their systems development, how do they reconcile the need for planning apparent in outsourcing with the need for agility? Also, to what extent do companies use RUP and/or follow reuse policies? These are just some of the many questions that can be answered only through field work in organizations that develop systems, either for sale or for their own use. Industry-based research will help identify best practices that can benefit many different organizations as they seek to improve their systems development efforts.

Finally, as Baskerville et al. (2009) suggested, we are on the verge of a new, fourth approach to systems development. What will this post-agility, post-plan-driven era look like? What should it look like? How can its advent be hastened? We are in a unique position vis-a-vis those who studied the transitions between the previous approaches to systems development—few of them anticipated the
previous transitions. Study of the transitions, and the new approaches they produced, came only after they had already occurred. Meanwhile, we know the next transition is coming, and we have some idea of what the next approach will look like. Not only can we anticipate the change and study it as it occurs, but we can also influence the form it eventually takes and its underlying philosophy.

28.7 Conclusions

Administrative information systems have been a part of organizational life for almost 60 years. During that time, information systems technology, organizations, the business environment, and the global economy have all changed dramatically. Information systems development methods have evolved to better deal with and take advantage of changes in technology as well as in the various external environments that affect it. Initially, when the technology was relatively primitive, systems development followed a craft approach. This was suitable for an era without standards, where computers varied considerably by vendor and by model, and where there were no standard programming languages, much less standard development approaches. Although the systems developed during this era worked, and some were truly innovative and worked amazingly well, the lack of predictability and documentation and the costs involved became too much to bear. Toward the end of this phase, standardized development methodologies were developed, just as standardized procedural programming languages had been developed. With standardization came control: over budgets, over schedules, over functionality, and over information systems personnel. Planning became paramount, and artists became engineers. While there was some resistance to this transformation, the plan-based approach soon became the norm. But in a dynamic human enterprise such as systems development, things did not stay the same for long. The plan-based approach had its own problems. To its critics, it was too slow to keep up with a dynamic business environment, it was too bloated, and it was too big. Such critiques were becoming common even before the Internet and the concept of “Internet time” exploded on the scene. Eventually, a new approach, agility, emerged. It was slow to spread, but eventually agility became just as popular as the plan-based approach. Today, these two approaches coexist, and there is some evidence to suggest that aspects of the two perspectives are slowly merging into a new approach. We can call this the “post-agility” phase, although a new and more descriptive name will be invented as the approach itself becomes better articulated. We know from the history of systems development approaches that the next phase will take some time yet to fully emerge. It may be 5 years from now, or it may be 10. If there is a revision of this chapter 10 years from now, the new phase will almost certainly be included. While there is still much uncertainty about the next phase, what it will encompass, and what it will be called, what is certain is that there will eventually be a fourth phase and that it will not be the last.

References


Further Reading


James, M. Available at http://www.jamesmartin.com.

UML. Available at http://www.uml.org.

Yourdon, E. Available at http://yourdon.com/about/.