Chapter 8
Software Design: Past and Present

A fact in itself is nothing. It is valuable only for the idea attached to it, or for the proof which it furnishes.

– Claude Bernard

8.1 A Software Design Framework

The design of software has been one of the greatest challenges in the development of information systems. From its fairly primitive beginnings in the form of toggling on/off switches and punching holes in paper tapes, software has come to dominate the cost of all forms of information systems. Yet, instead of gaining increasing mastery over the processes of software design, we continue to be challenged by new software technologies, greater quality expectations, and higher complexities of integrated systems. Thus, software design remains an essentially wicked problem that is typically crafted to each software-intensive system developed.

This chapter presents a brief overview of the progress made in software design over the past 60 years. The presentation is structured on the software design framework found in Fig. 8.1. We view software as composed of four basic design elements – an architecture; the algorithmic procedure in a programming language; the data in a structured format; and the human–computer interaction with enabling human–computer interfaces. Bringing these design elements together into an effective software design requires well-defined software development processes and development methods as shown at the center of the framework.

Our goal in this chapter is to present a brief historical retrospective of software design challenges and successful design solutions (Campbell-Kelly and Aspray 1996; Hevner and Berndt 2000). Learning from the lessons of the past we can hope to build improved software designs for present and future software-intensive systems.

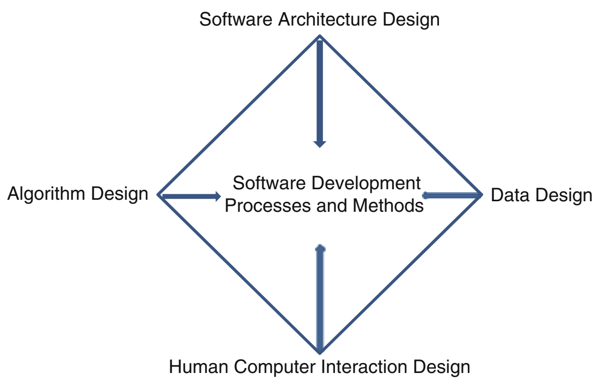
Software Architecture Design
Algorithm Design
Software Development Processes and Methods
Data Design
Human Computer Interaction Design

Fig. 8.1 Software design framework

8.2 Software Architecture

The three technology components of a computing environment – computing platform, communication networks, and software – are integrated via system architecture into a functional, effective information system. A classic paper by Zachman (1987) presents an information systems architecture framework made up of the elements of *data*, *process*, and *networking*. The Zachman IS architecture combines representations of these elements into an architectural blueprint for a business application. The framework of this chapter differs by including the computing platform (hardware and OS) as a fundamental systems element and combining algorithmic procedure (i.e., process) and data into the software architecture element.

We focus our discussion here on the software architecture component of the overall systems architecture. The goal of software architecture is to provide a mapping or "blueprint" to integrate all required functionalities and qualities of the desired information system provided by software. The objectives of all system stakeholders must be considered and represented in the design of the software architecture. Design trade-offs are key decision points in the development of an effective architecture.

The importance of software architectures for the development of complex systems has been clearly recognized in the software engineering literature (Shaw and Garlan 1996; Bass et al. 2003; Taylor et al. 2009). Architectural styles are identified based on their organization of software components and connectors for the transmission of data and control among components. The following discussion of software architectures is organized chronologically. Information systems have evolved and become more complex due to the requirements for meeting many, sometimes conflicting, functional, and quality objectives. We will see how over time the architectural solutions attempt to satisfy these multiple objectives.

8.2.1 Manual Business Processes

In the centuries leading up to the invention of the computer, businesses focused their creative energies on the development of effective business processes for production, personnel management, accounting, marketing, and sales. Standard operating procedures (SOPs) and workflow processes were widely used throughout business history. The concept of a "general systems theory" guided the structure and application of these business processes.

Even before the advent of computers, intellectual leaders, such as Herbert Simon and C. West Churchman, were extending the ideas of systems thinking into business organizations (Kast and Rosenzweig 1972). Such systemic business processes were performed manually up to around 1950. However, the business focus of getting the critical business processes right before automation remains an underlying tenet of all successful organizations today and for the foreseeable future.

8.2.2 Mainframe Architectures

The automation of business processes with the original large mainframe computer systems occurred slowly at first. The 1950s and early 1960s saw a vast majority of business application programs written in COBOL based on basic *data-flow architectures*. During this era computer systems consisted primarily of the computational platform (e.g., mainframe and operating system) and early application software systems. In a data-flow architecture, data in the form of variables, records, or files move from one computer system application to the next until the required business process is completed. The simplest form of a data-flow architecture is known as a *batch sequential architecture*. Data is batched into large files and the application programs are batched for sequential runs on the data files. The classic master file–transaction file applications are based on batch sequential processing. The *pipe and filter architecture* is a more general model of dataflow. Pipes carry data from one filter to the next in a network dataflow. A filter accepts streams of data as input, performs some processing on the data, and produces streams of data as output. The filter performs local transformations of an input into an output on a continuing basis. Each filter is independent of all other filters in the data-flow architecture. The pipe and filter structure provided the underlying computational model for the UNIX operating system (Bach 1987).

8.2.3 Online, Real-Time Architectures

During the time from 1965 to 1974, businesses began to realize the competitive advantages of online, real-time processing. The evolving technologies of databases, data communications, and the computational platform (e.g., minicomputers and real-time operating systems) enabled sophisticated real-time business and scientific applications to be developed. Online processing required important new advances in data communications (e.g., remote job entry, real-time data queries, and updates), database repositories (e.g., hierarchical and network databases), and operating systems (e.g., multiprogramming, real-time interrupts, resource allocation). The critical need was to align these new technologies and the software architecture with sufficient performance to meet rigorous response time and data capacity requirements.

The principal architecture used to meet these requirements was a *repository architecture*. A central repository of operational data in file and database formats represents the current state of the application system. Multiple sources of independent transactions (i.e., agents) perform operations on the repository. The interactions between the central repository and the external agents can vary in complexity, but in the early days of online applications they consisted mostly of simple queries or updates against the repository. The real-time operating system provides integrity and concurrency control as multiple transactions attempt to access the data in real time. The data-centric nature of most business applications has made the repository architecture with real-time requirements a staple of application development.

8.2.4 Distributed, Client–Server Architectures

Around 1975 decentralization of control became a key business and information systems strategy. The ability to decentralize the organization and move processing closer to the customer brought about major changes in thinking about work processes and the supporting computer systems. The technology components to support true distributed processing were available during this era to support these decentralized business strategies. Networks of communicating computers consisted of mainframes, minicomputers, and increasingly popular microcomputers. *Distributed architectures* became the norm for building new business computer systems (Peebles and Manning 1978; Scherr 1978).

Distributed computing provided a number of important advantages for computing systems. Partitioning the workload among several processors at different locations enhanced performance. System availability was increased due to redundancy of hardware, software, and data in the system. Response time to customer requests was improved since customer information was located closer to the customer site. The ability to integrate minicomputers and microcomputers into the distributed architecture provided significant price-performance advantages. The potential disadvantages of the distributed system were loss of centralized control of data and applications and performance costs of updating redundant data across the network.

An important variant of the distributed architecture is the *client–server architecture*. A server process, typically installed in a larger computer, provides services to client processes, typically distributed on a network. A server can be independent of the number and type of its clients, while a client must know the identity of the server and the correct calling sequence to obtain service. Examples of services include database systems and specialized front-end and back-end components.

8.2.5 Component-Based Architectures

During the latter decades of the 20th century, an important focus was on the alignment of business strategy with information technology (IT) strategy in the organization. A *strategic alignment model* proposed by Henderson and Venkatraman (1993) posits four basic alignment perspectives:

1. Strategy execution: The organization's business strategy is well defined and determines the organizational infrastructure and the IT infrastructure. This is the most common alignment perspective.
2. Technology transformation: The business strategy is again well defined, but in this perspective it drives the organization's IT strategy. Thus, the strategies are in alignment before the IT infrastructure is implemented.
3. Competitive potential: The organization's IT strategy is well defined based upon innovative IT usage to gain competitive advantage in the marketplace. The IT strategy drives the business strategy which in turn determines the organizational infrastructure. The strategies are aligned to take advantage of the IT strengths of the organization.
4. Service level: The IT strategy is well defined and drives the implementation of the IT infrastructure. The organizational infrastructure is formed around the IT infrastructure. The business strategy does not directly impact the IT strategy.

While all four perspectives have distinct pros and cons, the alignment of the business strategy and the IT strategy before the development of the organizational and IT infrastructures in perspectives 2 and 3 provides a consistent vision to the organization's business objectives. This vision was translated into the system and software architectures of the organization's IT systems.

This time period saw the traditional business strategy of "make and sell" transformed into a strategy of "sense and respond" (Haeckel and Nolan 1996). Two new software architectures were devised to meet these changing environmental demands.

Event-driven architectures have become prevalent in business systems that must react to events that occur in the business environment (Barrett et al. 1996). When an important event occurs a signal is broadcast by the originating component. Other components in the system that have registered an interest in the event are notified and perform appropriate actions. This architecture clearly performs well in a "sense and respond" business environment. Note that announcers of events are unaware of which other components are notified and what actions are generated by the event. Thus, this architecture supports implicit invocation of activity in the system. This architecture provides great flexibility in that actions can be added, deleted, or changed easily for a given event.

The important new development ideas of component-based development have led naturally to the design and implementation of *component-based architectures* (Brown 2000). Business systems are composed of functional components glued together by middleware standards such as CORBA, DCOM, and Enterprise JavaBeans. In many cases the components are commercial off-the-shelf (COTS) products. Thus, organizations are able to build complex, high-performance systems by integrating COTS components via industry standard middleware protocols. This minimizes development risk while allowing the business organization to effectively align its IT strategy with its business strategy via judicious selection of best practice functional components. Enterprise resource planning (ERP) business systems from vendors like SAP, Baan, and Oracle utilize component-based architectures to allow clients to customize their business systems to their organization's requirements.

8.2.6 Service-Oriented Architectures

The influence of the World Wide Web has required businesses to rethink their business and IT strategies to take greatest advantage of its revolutionary impact. This current era of Internet and ubiquitous computing will generate new web-based *service-oriented architectures* (SOA) for integrating the capabilities of the Internet into business functions of marketing, sales, distribution, and funds transfer (Erl 2005). Service orientation provides a loose coupling of services glued together in business flows to support end-user goals of functionalities and qualities. SOA separates these functions and qualities into distinct units, or services, which developers make accessible over a network in order that users can combine and reuse them in the production of applications. These services communicate with each other by passing data from one service to another or by coordinating an activity between two or more services.

The implications of service-oriented architectures are just now being studied. The rapid exchange of information via push and pull technologies to any point of the globe will eliminate most boundaries and constraints on international commerce. However, critical issues of security, privacy, cultural differences (e.g., language), intellectual property, and political sensitivities will take many years to be resolved.

8.3 Algorithmic Design

Solving a scientific or business problem first requires the creation of an algorithm that provides a step-by-step procedure for accepting inputs, processing data, and producing outputs. Algorithms are typically represented in natural language or some form of structured format like pseudocode or flowcharts. The objective of computer programming is to code this algorithm in a form such that an important problem can be solved via the use of a computer system. The history of computer programming languages has been documented in several excellent sources (Wexelblat 1981; Bergin and Gibson 1996) and will not be addressed here.

8.3.1 Early Program Design

Programming in the early days of computing involved wiring plugboards, setting toggle switches on the side of the computer, or punching holes in paper tape. The wiring, switch settings, and holes represented instructions that were interpreted and executed by the computer. Each time a program was run the boards had to be re-wired, switches had to be reset, or the paper tape re-punched. The stored program computer changed this onerous task by storing the program in internal memory and executing it upon command.

The 1960s brought major advances in program compilers and assemblers. The role of a compiler is to translate a high-level language in which humans can write algorithms into a computer's internal set of instructions. The assembler then translates the computer instructions into binary machine code for placement in the computer's memory. The research and development of compilers and assemblers led rapidly to the creation of the first computer programming languages.

IBM developed FORTRAN (FORmula TRANslation) for the 704 computer in early 1957, contributing to the popularity of the product line. John Backus and his team defined the language to support engineering applications that required fast execution of mathematical algorithms (Backus 1979). FORTRAN has gone through many generations and remains popular even today for engineering applications.

Business computing had different requirements. Efficient data handling was critical and business programmers had the need for a more user-friendly, English-like language. In 1959, a team of Department of Defense developers, including Grace Murray Hopper, defined a business-oriented programming language COBOL (common business-oriented language). COBOL is a highly structured, verbose language with well-defined file handling facilities. The English-like syntax makes programs more readable and self-documenting. It was also the first language to be standardized, so its programs could run on different hardware platforms.

8.3.2 Structured Program Design

As application software systems grew in size a crisis in software development was created. Large programs (e.g., 50,000–100,000 lines of code) were being developed. These programs were very difficult to read, debug, and maintain. Software routinely failed and repairs were difficult and time-consuming. The worldwide nature of the software problem was reflected in the NATO software engineering conferences held in 1968 (Garmisch, Germany) and 1969 (Rome) (Naur and Randell 1969; Buxton and Randell 1970). The term "software engineering" was coined to generate discussion as to whether the development of software was truly an engineering discipline.

The issues of software development and how to solve the software crisis debated at these early conferences remain relevant even today.

Edsger Dijkstra's influential 1968 paper, "Go-To Statement Considered Harmful," (Dijkstra 1968) addressed a major problem in existing programming languages. Flow of logical control through a program was often haphazard leading to "spaghetti code" programs. Software researchers, like Dijkstra and Harlan Mills, proposed structured programming as the answer to out-of-control program flow (Mills 1986). The use of only three simple structures – sequence, selection, and iteration – can express the control flow of any algorithm (Boehm and Jacopini 1966). This understanding led to the development of new structured programming languages.

The languages Pascal and ALGOL-68 initiated some of the principal structured programming concepts. However, they had little commercial impact. Meanwhile, new versions of FORTRAN-IV and COBOL integrated new structured features.

8.3.3 Recent Algorithm Design Paradigms

Object-oriented (OO) design originated in simulation languages, such as Simula, during the 1960s. An object-oriented design of an algorithm models a collection of cooperating *objects*. Each object bounds a coherent set of activities (methods) and information (data structures). An object can be viewed as an independent entity in the application world with capabilities of interacting with other objects via the receiving and sending of messages. Object-oriented design highlights the design principles information hiding, data abstraction, encapsulation, modularity, polymorphism, and inheritance (Booch, Maksimchuk, Engel, Young, Conallen, and Houston 2007). Many current programming languages support OO design and coding.

Aspect-oriented (AO) design extends the OO paradigm by identifying cross-cutting concerns (e.g., aspects) in an application design (Jacobson and Ng 2005). By separating distinct concerns, design modularity is improved and satisfaction of the cross-cutting aspect can be more effectively designed, implemented, tested, and monitored in operation. Examples of aspects are security, logging, and transactions. Research and development on aspect-oriented design and programming is still in early stages.

8.3.4 Widely Used Programming Languages

In the early 1970s, Ken Thompson and Dennis Ritchie developed a new systems programming language called C, using the language to implement UNIX. By allowing access to low-level machine operations, this language defied many of the tenets of structured programming. Regardless, C has become a very popular programming language, particularly within the last two decades of programming as personal computers have dominated the desktops. The language C++ evolved from C for the programming of object-oriented business applications (Ritchie and Thompson 1974). Variations of the C language proliferate in today's programming environments.

Visual programming languages, such as Visual Basic (VB), incorporate facilities to develop graphic user interfaces (GUIs) for business applications. Such languages are particularly effective in the development of client–server distributed applications where the end-user at the client site needs an efficient, friendly interface.

The advent of the Internet and service-oriented architectures has provided the impetus for efficient, platform independent programs that can be distributed rapidly to Internet sites and executed. The Java programming language was developed at Sun Microsystems to fit this need (Gosling et al. 2005).

8.4 Data Design

The management of data in systems predates recorded history. The first known writing was done on Sumerian stone tablets and consisted of a collection of data on royal assets and taxes. Writing on papyrus and eventually on paper was the predominate manner of manual data management up to the beginning of the 20th century. First mechanical and then electronic machinery rapidly changed the ways in which data is managed.

8.4.1 Punched Card Data Management

Although automated looms and player pianos used punched cards to hold information, the first true automated data manager was the punched card system designed by Herman Hollerith to produce the 1890 census. Automated equipment for handling and storing punched cards was the primary means of managing business data until the era of automation. An entire data management industry, whose leader was IBM, grew up around punched cards.

8.4.2 Computerized File Management

The use of the UNIVAC I computer system for the 1950 census heralded the era of automation in business computing. To replace punched cards, magnetic drums and tapes were developed to store data records. Without the constraints of an 80-column card format, new, innovative data structures were devised to organize information for fast retrieval and processing. Common business applications for general-ledger, payroll, banking, inventory, accounts receivable, shipping invoices, contact management, human resources, etc., were developed in COBOL. All of these programs were centered on the handling of large files of data records. The prevailing system architecture during this era was that of batch-oriented processing of individual transactions. Files of transactions were run against a master file of data records, usually once a day or once a week. The problems with this architecture were the inherent delays of finding and correcting errors in the transactions and the lack of up-to-date information in the master file at any point in time (Gray 1996).

8.4.3 Online Data Processing

Direct access to data in magnetic storage led to improved structures for rapidly locating a desired data record in a large data file while still allowing efficient sequential processing of all records. *Hierarchical data models* presented data in hierarchies of one-to-many relationships. For example, a department record is related to many employee records and an employee record is related to many project records. Sophisticated indexing techniques provided efficient access to individual data records in the file structure. Popular commercial file systems, such as IBM's indexed sequential access mechanism (ISAM) and virtual sequential access mechanism (VSAM), provided very effective support for complex business applications based on large data sets.

Hierarchical data models lacked a desired flexibility for querying data in different ways. In the above example, the hierarchy as designed would not efficiently support the production of a report listing all employees working on a given project. A more general method of modeling data was needed. An industrial consortium formed the Data Base Task Group (DBTG) to develop a standard data model. Led by Charles Bachman, who had performed research and development of data models at General Electric, the group proposed a *network data model*. The DBTG network model was based on the insightful concepts of data independence and three levels of data schemas:

- External subschema: Each business application had its own subset view of the complete database schema. The application subschema was optimized for efficient processing.
- Conceptual schema: The global database schema represented the logical design of all data entities and the relationships among the entities.
- Physical schema: This schema described the mapping of the conceptual schema onto the physical storage devices. File organizations and indexes were constructed to support application processing requirements.

Data independence between the schema levels allowed a developer to work at a higher level while remaining independent of the details at lower levels. This was a major intellectual advance that allowed many different applications, with different subschemas, to run on a single common database platform.

8.4.4 Relational Databases

E.F. Codd, working at IBM Research Laboratory, proposed a simpler way of viewing data based on relational mathematics (Codd 1970). Two-dimensional relations are used to model both data entities and the relationships among entities based upon the matching of common attribute values. The mathematical underpinnings of the *relational data model* provided formal methods of relational calculus and relational algebra for the manipulation and querying of the relations. A standard data definition and query language, structured query language (SQL), was developed from these foundations.

Commercialization of the relational model was a painstaking process. Issues of performance and scalability offset the advantages of easier conceptual modeling and standard SQL programming. Advances in query optimization led to relational systems that could meet reasonable performance goals. The relational model fit nicely with new client–server architectures. The move of processing power to distributed client sites called for more user-friendly graphical user interfaces and end-user query capabilities. At the same time, more powerful processors for the servers boosted performance for relational processing.

8.4.5 Current Trends in Data Management

Data design and the effective management of data have always been and will remain the center of most computing systems. The digital revolution has drastically expanded our definition and understanding of knowledge, information, and data. Multimedia data includes audio, pictures, video, documents, touch (e.g., virtual reality), and, maybe even, smell. New applications are finding effective ways of managing and using multimedia data.

Object-oriented methods of software development attempt to break the boundary between algorithmic procedures and data (Stonebraker 1996). Two main themes characterize the use of object technology in database management systems: object-relational technology integrates the relational model and support for objects, while object-oriented systems take a more purist approach.

A major challenge for the future of data management will be how to manage the huge amounts of information flowing over the World Wide Web. It is estimated that a majority of business (e.g., marketing, sales, distribution, and service) will be conducted over the Internet in the near future. New structures for web databases must support real-time data capture, ongoing analyses of data trends and anomalies (e.g., data mining), multimedia data, real-time data streaming, and high levels of security. In addition, web-enabled business will require very large databases, huge numbers of simultaneous users, and new ways to manage transactions.

8.5 Human–Computer Interaction (HCI) Design

The effectiveness of any computer application is determined by the quality of its interactions and interfaces with the external world. This area of research and development has been termed human–computer interaction (HCI).

8.5.1 Early Computer Interactions

The first computer interactions were via toggle switches, blinking lights, paper tape punches, and primitive cathode-ray tubes. Quickly the need for more effective input/output devices brought about the use of card readers/punches and teletype printers. Up to 1950, however, interaction with the computer was the domain of computer specialists who were trained to handle these arcane and unwieldy interfaces.

The use of computers in effective systems required more usable, standard human–computer interfaces. Early on, the standard input medium was the Hollerith card. Both the program and data were keypunched on cards in standardized formats. The cards were organized into card decks, batched with other card decks, and read into the computer memory for execution. Output was printed on oversized, fan-fold computer paper. Businesses were required to hire computer operations staff to maintain the computer systems and to control access to the computer interfaces. End-user computing was rare during this era.

8.5.2 Text-Based Command Interfaces

As computer use grew during the 1970s, the demand from end-users for more effective, direct interaction with the applications grew correspondingly. Moving from batch computer architectures to online distributed architectures necessitated new terminal-based interfaces for the application users. Computer terminals were designed to combine a typewriter input interface with a cathode-ray tube output interface. Terminals were connected to the mainframe computer via direct communication lines. The design of the computer terminal was amazingly successful and remains with us today as the primary HCI device. HCI interfaces for online applications were either based on scrolling lines of text or on predefined bit-mapped forms with fields for text or data entry.

Standard applications began to proliferate in the environment of online computing, for example

- Text editing and word processing: The creation, storage, and manipulation of textual documents rapidly became a dominant use of business computers. Early text editors were developed at Stanford, MIT, and Xerox PARC. Commercial WYSIWYG (what you see is what you get) word processing packages came along in the early 1980s with LisaWrite, a predecessor to MacWrite, and WordStar.

- Spreadsheets: Accounting applications are cornerstone business activities. Commercial accounting packages have been available for computers since the 1950s. The spreadsheet package VisiCalc became a breakthrough product for business computing when it was introduced in 1979. Lotus 1-2-3 and Microsoft Excel followed as successful spreadsheet packages.
- Computer-aided design: The use of computers for computer-aided design (CAD) and computer-aided manufacturing (CAM) began during the 1960s and continues today with sophisticated application packages.
- Presentation and graphics: Research and development on drawing programs began with the sketchpad system of Ivan Sutherland in 1963. Computer graphics and paint programs have been integrated into business applications via presentation packages, such as Microsoft's PowerPoint.

Text-based command languages were the principal forms of HCI during the 1970s and 1980s for the majority of operating systems, such as UNIX, IBM's MVS and CICS, and DEC's VAX VMS. The users of these systems required a thorough knowledge of many system commands and formats. This type of text-based command language carried over to the first operating systems for personal computers. CPM and MS-DOS constrained users to a small set of pre-defined commands that frustrated end-users and limited widespread use of personal computers.

8.5.3 The WIMP Interface

Many years of research and development on computer graphical user interfaces (GUIs) have led to today's WIMP (windows, icons, mouse, and pull-down menus) HCI standards. Seminal research and development by J. Licklider at ARPA, Douglas Englebart at Stanford, and the renowned group at Xerox PARC led to the many innovative ideas found in the WIMP interface (Myers 1998). The first commercial computer systems popularizing WIMP features were the Xerox Star, the Apple Lisa, and the Apple Macintosh. The X Window system and the Microsoft Windows versions made the WIMP interface a standard for current computing systems.

More than any other technology, the WIMP interface and its ease of use brought the personal computer into the home and made computing accessible to everyone. Advantages to businesses included an increase in computer literate employees, standard application interfaces across the organization, decreased training time for new applications, and a more productive workforce.

8.5.4 Current Trends in HCI

As with all computer technologies the Internet has brought many changes and new challenges to HCI. The World Wide Web is based on the concept of hypertext whereby documents are linked to related documents in efficient ways. Documents on the Internet use a standard coding scheme (HTML and URLs) to identify the locations of the linked documents. Specialized web browsers provide the interfaces for viewing documents on the web. Mosaic from the University of Illinois was the first popular web browser. Currently, open source Apache and Microsoft Internet Exchange (IE) provide the most widely used web browsers. New information exchange standards, such as the Extensible Markup Language (XML), will support improved methods for moving both data and metadata (e.g., semantics) on the WWW.

There are numerous important new directions in the field of HCI, for example

- Gesture recognition: The recognition of human gestures began with light pens and touch-sensitive screens. Hand writing recording and recognition is a subject of ongoing research.
- Three-dimensional graphics: Research and development on three-dimensional interfaces has been an active area, particularly in CAD-CAM systems. Three-dimensional visualization of the human body has the potential to revolutionize surgery and health care.
- Virtual reality: Scientific and business uses of virtual reality are just now being explored. Head-mounted displays and data gloves will become commercially viable in the near future for marketing demonstrations and virtual design walkthroughs.
- Voice recognition and speech: audio interfaces to computer systems have been available for the past decade. However, the limited vocabulary and requirements for specific voice pattern recognition remain problems to overcome before widespread use.
- Mobile devices: HCI and effective interfaces for mobile devices call for a full understanding of the challenges (e.g., battery power, small screen size) and opportunities (e.g., connectivity, computing power) found in handheld and wearable devices.

8.6 Software Development Processes and Methods

The four software design components of software architecture, algorithmic programming, data, and HCI are brought together in the design and implementation of a business application via software development processes and methods. A *software development process* is a pattern of activities, practices, and transformations that support managers and engineers in the use of technology to development and maintain software systems. A *software development method* is a set of principles, models, and techniques for effectively creating software artifacts at different stages of development (e.g., requirements, design, implementation, testing, and deployment). Thus, the process dictates the order of development phases and the transition criteria for transitioning from one phase to the next, while the method defines what is to be done in each phase and how the artifacts of the phase are represented. The history of software design and development has seen important advances in both software processes and software methods. We briefly track the evolution of these advances in this section.

8.6.1 Software Development Processes

In early software development projects very little attention was paid to organizing the development of software systems into stages. Programmers were given a problem to solve and were expected to program the solution for computer execution. The process was essentially "code and fix." As problems became more complex and the software grew in size this approach was no longer feasible.

The basic "waterfall process" was defined around 1970 (Royce 1970). A well-defined set of successive development stages (e.g., requirements analysis, detailed design, coding, testing, implementation, and operations) provided enhanced management control of the software development project. Each stage had strict entrance and exit criteria. Although it had several conceptual weaknesses, such as limited feedback loops and overly demanding documentation requirements, the waterfall process model served the industry well for over 20 years into the 1990s. The principal department of defense process standard for the development of system software systems during this period, DOD-STD-2167A, was based on the waterfall approach.

Innovative ideas for modeling software development processes include the *spiral model* (Boehm 1988) and *incremental development* (Trammell et al. 1996). The spiral model shows the development project as a series of spiraling activity loops. Each loop contains steps of objective setting, risk management, development/verification, and planning. Incremental development emphasizes the importance of building the software system in well-defined and well-planned increments. Each increment is implemented and certified correct before the next increment is started. The system is thus grown in increments under intellectual and management control.

A standard, flexible software development process is essential for management control of development projects. Recent efforts to evaluate the quality of software development organizations have focused on the software development process. The Software Engineering Institute (SEI) proposed the capability maturity model (CMM) to assess the maturity of an organization based on how well key process areas are performed (Paulk 1994). The CMM rates five levels of process maturity:

- Initial: Ad hoc process
- Repeatable: Stable process with a repeatable level of control
- Defined: Effective process with a foundation for major and continuing progress
- Managed: Mature process containing substantial quality improvements
- Optimized: Optimized process customized for each development project

The principal goal of the CMM was for organizations to understand their current process maturity and to work toward continuous process improvement. The international ISO-9000 standards contain similar provisions to evaluate the effectiveness of the process in software development organizations.

8.6.2 Early Development Methods

Early methods of program design were essentially ad hoc sketches of logic flow leading to the primary task of writing machine code. These design sketches evolved into flowcharting methods for designing program logic. Basic techniques also evolved for the development activities of requirements analysis, software design, and program testing.

The creation of software was initially considered more of a creative art than a science. As computing systems and software requirements became more complex into the 1960s, development organizations quickly lost the ability to manage software development in a predictable way. Defined development processes and methods brought some controls to software construction. Structured methods for the analysis and design of software systems appeared during the late 1960s and early 1970s. Two primary approaches were defined – *procedure-oriented methods* and *data-oriented methods*.

The development of procedure-oriented methods was strongly influenced by the sequential flow of computation supported by the dominant programming languages, COBOL and FORTRAN. The focus of software development under this paradigm is to identify the principal functions (i.e., procedures) of the business system and the dataflows among these functions. The system functions are hierarchically decomposed into more detailed descriptions of subfunctions and dataflows. After sufficient description and analysis, the resulting functions and data stores are designed and implemented as software modules with input–output interfaces. Primary examples of procedure-oriented system development methods include structured analysis and structured design methods (Stevens et al. 1974; Yourdon 1989).

Data-oriented system development places the focus on the required data. The data-centric paradigm is based on the importance of data files and databases in large business applications. System data models are developed and analyzed. System procedures are designed to support the data processing needs of the application. The design and implementation of the application software is constructed around the database and file systems. Primary data-oriented methods included the Warnier–Orr methods (Orr 1977) and information engineering (Martin 1989).

8.6.3 Object-Oriented Methods

In the early 1980s, object-oriented (OO) methods of software development were proposed for building complex software systems. Object orientation is a fundamentally different view of a system as a set of perceptible objects and the relationships among the objects. Each object in the application domain has a *state*, a set of *behaviors*, and an *identity*. A business enterprise can be viewed as a set of persistent objects. Business applications are developed by designing the relationships and interactions among the objects. Advocates point out several significant advantages of OO system development, including increased control of enterprise data, support for reuse, and enhanced adaptability to system change. Risks of OO development include the potential for degraded system performance and the startup costs of training and gaining OO experience. A plethora of OO development methods in the 1980s have converged into the unified modeling language (UML) standards (Fowler 2003).

8.6.4 Formal Development Methods

The requirement for highly reliable, safety-critical systems in business, industry, and the public sector has increased interest in formal software development methods. Formal methods are based on rigorous, mathematics-based theories of system behavior (Wing 1990; Luqi and Goguen 1997). Formal methods, such as the cleanroom methods (Mills et al. 1986; Prowell et al. 1999), support greater levels of correctness verification on all artifacts in the development process: requirements, design, implementation, and testing. The use of formal methods requires the use of mathematical representations and analysis techniques entailing significant discipline and training in the development team. While anecdotal evidence of the positive effects (e.g., improved quality and increased productivity) of formal methods is often reported (Gerhart et al. 1993) more careful study of the trade-offs in implementing formal methods are needed (Pfleeger and Hatton 1997).

8.6.5 Component-Based Development (CBD) Methods

The current widespread trend for the development of large-scale computing systems is component-based development (CBD). CBD extends the ideas of software reuse into a full-scale development process whereby complete business applications are delivered based upon the interaction and integration of software components (Brown 2000). A component is essentially a software module with well-defined interfaces to the external world. Thus, each component provides a service to the application system. New products and standards for middleware provide the "glue" for building systems from individual components. The technologies of DCOM, CORBA, and Enterprise JavaBeans are a start for enabling CBD processes and methods. Object-oriented concepts, such as encapsulation and class libraries, and emphasis on system architectures, such as *n*-tier client–server, support the realization of CBD in application environments.

8.6.6 Agile Development Methods

Agile software development espouses a philosophy of building software systems where requirements and working software evolve through interactions among self-organizing, cross-functional developer teams. The Agile Manifesto (2001), as developed by a core group of software thinkers, promotes the following set of principles:

"We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

That is, while there is value in the items on the right, we value the items on the left more."

A number of popular software development approaches, such as extreme programming (XP) (Beck and Andres 2005) and Scrum (Schwaber 2004), are based on the philosophy and practices of the agile movement.

8.6.7 Controlled-Flexible Development Methods

A central dynamic in software development is the trade-off between control and flexibility. Strong management control is desirable in environments with strict budgetary or schedule constraints and those that require high-quality results, such as safety-critical systems. This control is typically achieved through upfront development of a plan that defines system requirements, architectures, designs, budgets, and schedules with the hope that the development will proceed smoothly based on the pre-determined script. In executing the plan, the developers have limited flexibility to modify their activities within the plan's controls.

A movement to encourage flexibility in software projects has resulted in a number of development approaches, such as rapid prototyping (Baskerville and Stage 1996), synchronize and stabilize (Cusumano and Yoffie 1999), the rational unified process (RUP) (Kruchten 2000), and the agile approaches of XP and Scrum as mentioned in the previous section. The essence of flexibility is the ability to improvise in reaction to changes – changes in requirements, budgets, schedules, risks, etc.

The goal is to find the most effective trade-off between control and flexibility for each specific software development project. Boehm and Turner (2004) propose using project dimensions such as technology sophistication, project size, staffing expertise, diversity of stakeholders, and application domains. While others highlight the need to tailor development methods to projects (Fitzgerald et al. 2003; Fitzgerald et al. 2006). However, there is a scarcity of theory and practical guidance to describe the balance between control and flexibility for a software development project.

Recent work by Harris et al. (2009a, 2009b) proposes the use of emergent controls such as *scope boundaries* and *ongoing feedback* to understand the trade-offs between control and flexibility. Scope boundaries constrain the set of feasible solutions without dictating specific outcomes. These boundaries can be thought of as risk management mechanisms that shape the attention of the software team. Examples of scope boundaries from XP include a shared vision, partial specifications, predefined architectures, fixed APIs, and resource constraints. The intersection of all the scope boundary controls defines the feasible space for exploration.

Ongoing feedback in software development projects provide teams with checkpoints to validate that progress is headed in the right direction based on feedback from multiple stakeholders, including all members of the development team. Teams with few scope boundaries need more feedback.

For example, XP methods have broad boundaries but contain pervasive feedback techniques including pair programming, daily team review meetings, co-location of team members with visible progress indicators, daily software builds, co-location with customer representatives, and very short release cycles to gain broad market exposure. In contrast, the rational unified process (RUP) approach tightly defines development scope and only offers feedback through iteration releases. If scope boundaries are tightened sufficiently to remove all choice from the development team, there is no need for interim feedback and the approach becomes more of a specification-driven approach. Thus, the goal of a controlled-flexible development method is to achieve a trade-off among emergent outcome controls, balancing the restrictiveness of scope boundaries with opportunities for dynamic feedback.

References

Agile Manifesto (2001) *Manifesto for Agile Software Development*, http://agilemanifesto.org.

Bach, M. (1987) *The Design of the UNIX Operating System*, Prentice Hall, Englewood Cliffs, NJ.

Backus, J. (1979) The history of FORTRAN I, II, and III, *IEEE Annuals of the History of Computing* 1 (1), pp. 21–37.

Barrett, D., L. Clarke, P. Tarr, and A. Wise (1996) A framework for event-based software integration, *ACM Transactions on Software Engineering and Methodolog*y 5 (4), pp. 378–421.

Baskerville, R. and J. Stage (1996) Controlling prototype development through risk analysis, *MIS Quarterly* 20 (4), pp. 481–504.

Bass, L., P. Clements, and R. Kazman (2003) *Software Architecture in Practice*, 2nd edn, Addison-Wesley, Reading, MA.

Beck, K. and C. Andres (2005) *Extreme Programming Explained: Embrace Change, 2nd ed, XP Series*, Addison-Wesley, Inc., Boston, MA.

Bergin, T. and R. Gibson (eds) (1996) *History of Programming Languages II*, ACM Press, Addison-Wesley, Reading, MA.

Boehm, B. (1988) A spiral model of software development and enhancement, *IEEE Computer* 21 (5), pp. 61–72.

Boehm, B. and R. Turner (2004) *Balancing Agility and Discipline: A Guide for the Perplexed*, Addison-Wesley, Inc., Boston, MA.

Boehm, C. and G. Jacopini (1966) Flow diagrams, turing machines, and languages with only two formation rules, *Communications of the ACM* 9 (5), pp. 366–371.

Booch, G., R. Maksimchuk, M. Engel, B. Young, J. Conallen, and K. Houston, (2007) *Object-Oriented Analysis and Design with Applications*, 3rd edn, Addison-Wesley, Inc., Boston, MA.

Brown, A. (2000) *Large Scale Component Based Development*, Prentice-Hall PTR, Upper Saddle River, NJ.

Buxton, J. and B. Randell (1970) *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee*, Rome, Italy, 1969, NATO.

Campbell-Kelly, M. and W. Aspray (1996) *Computer: A History of the Information Machine*, Basic Books, New York.

Codd, E. (1970) A relational model of data for large shared data banks, *Communications of the ACM* 13 (6), pp. 377–387.

Cusumano, M. and D. Yoffie (1999) Software development on Internet time, *IEEE Computer* 32 (10), pp. 60–69.

Dijkstra, E. (1968) The Go-To statement considered harmful, C*ommunications of the ACM* 11 (3), pp. 147–148.

Erl, T. (2005) *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*, Prentice-Hall PTR, Upper Saddle River, NJ.

Fitzgerald, B., N. Russo, and T. O'Kane (2003) Software development tailoring at Motorola, *Communications of the ACM* 46 (4), pp. 65–70.

Fitzgerald, B., G. Hartnett, and K. Conboy (2006) Customizing agile methods to software practices at Intel Shannon, *European Journal of Information Systems* 15 (2), pp. 200–213.

Fowler, M. (2003) *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3rd edn, Addison-Wesley, Inc., Boston, MA.

Gerhart, S., D. Craigen, and A. Ralston (1993) Observation on industrial practice using formal methods, *Proceedings of the 15th International Conference on Software Engineering*, Computer Society Press, Silver Spring, MD.

Gosling, J., B. Joy, G. Steele, and G. Bracha (2005) *The Java Language Specification*, 3rd edn, Addison-Wesley, Boston, MA.

Gray, J. (1996) Evolution of data management, *IEEE Computer* 29 (10), pp. 38–46.

Haeckel, S. and R. Nolan (1996) Managing by Wire: Using IT to Transform a Business from "Make and Sell" to "Sense and Respond", Chapter 7 in J. Luftman (ed.) *Competing in the Information Age: Strategic Alignment in Practice*, Oxford University Press, Oxford.

Harris, M., A. Hevner, and R. Collins (2009a) Controls in flexible software development, *Communications of the Association for Information Systems* 24 (43), pp. 757–776.

Harris, M., R. Collins, and A. Hevner (2009b) Control of flexible software development under uncertainty, *Information Systems Research: Special Issue on Flexible and Distributed Information Systems Development* 20 (3), pp. 400–419.

Henderson, J. and N. Venkatraman (1993) Strategic alignment: leveraging information technology for transforming organizations," *IBM Systems Journal* 32 (1), pp. 4–16.

Hevner, A. and D. Berndt (2000) Eras of Business Computing, in M. Zelkowitz (ed.) *Advances in Computers, Vol. 52*, Academic Press, Inc., San Diego, CA, pp. 1–90.

Jacobson, I. and P. Ng (2005) *Aspect-Oriented Software Development with Use Cases*, Addison-Wesley, Inc., Boston, MA.

Kast, F. and J. Rosenzweig (1972) General systems theory: applications for organization and management, *Academy of Management Journal* 15 (3), pp. 447–465.

Kruchten, P. (2000) *The Rational Unified Process: An Introduction*, 2nd edn, Addison-Wesley, Inc., Reading, MA.

Luqi and J. Goguen (1997) Formal methods: promises and problems, *IEEE Software* 14 (1), pp. 73–85.

Martin, J. (1989) *Information Engineering: Books 1-3*, Prentice-Hall, Englewood Cliffs, NJ.

Mills, H. (1986) Structured programming: retrospect and prospect, *IEEE Software* 3 (6), pp. 58–66.

Mills, H., R. Linger, and A. Hevner (1986) *Principles of Information Systems Analysis and Design*, Academic Press, Inc., Boston, MA.

Myers, B. (1998) A brief history of human-computer interaction technology, *Interactions* 5 (2), pp. 44–54.

Naur, P. and B. Randell (1969) *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee*, Garmisch, Germany, 1968, NATO.

Orr, K. (1977) *Structured Systems Development*, Yourdon Press, Prentice-Hall, Englewood Cliffs, NJ.

Paulk, M. (ed.) (1994) *The Capability Maturity Model: Guidelines for Improving the Software Process*, Addison-Wesley, Reading, MA.

Peebles, R. and E. Manning (1978) System architecture for distributed data management, *IEEE Computer* 11 (1), pp. 40–47.

Pfleeger, S. and L. Hatton (1997) Investigating the influence of formal methods, *IEEE Computer* 30 (2), pp. 33–43.

Prowell, S., C. Trammell, R. Linger, and J. Poore (1999) *Cleanroom Software Engineering: Technology and Process*, Addison-Wesley, Inc., Reading, MA.

Ritchie, D. and K. Thompson (1974) The UNIX time-sharing system, *Communications of the ACM* 17 (7), pp. 365–375.

Royce, W. (1970) Managing the development of large software systems: concepts and techniques, *Proceedings of IEEE WESTCON*, Los Angeles.

Scherr, A. (1978) Distributed data processing, *IBM Systems Journal* 17 (4), pp. 324–343.

Schwaber, K. (2004) *Agile Project Management with Scrum*, Microsoft Press, Inc, Redmond, WA.

Shaw, M. and D. Garlan (1996) *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, Englewood Cliffs, NJ.

Stevens, W., G. Myers, and L. Constantine (1974) Structured design, *IBM Systems Journal* 13 (2), pp. 115–139.

Stonebraker, M. (1996) *Object-Relational DBMSs: The Next Great Wave*, Morgan Kaufmann, San Francisco, CA.

Taylor, R., N. Medvidovic, and E. Dashofy (2009) *Software Architecture: Foundations, Theory and Practice*, John Wiley & Sons, Hoboken, NJ.

Trammell, C., M. Pleszkoch, R. Linger, and A. Hevner (1996) The incremental development process in cleanroom software engineering, *Decision Support Systems* 17 (1), pp. 55–71.

Wexelblat, R. (ed.) (1981) *History of Programming Languages I*, Academic Press, Boston, MA.

Wing, J. (1990) A Specifier's introduction to formal methods, *IEEE Computer* 23 (9), pp. 10–23.

Yourdon, E. (1989) *Modern Structured Analysis*, Yourdon Press, Prentice-Hall, Englewood Cliffs, NJ.

Zachman, J. (1987) A framework for information systems architecture," *IBM Systems Journal* 26 (3), pp. 276–292.