**Handouts of Lecture 27 Professional Practices (IT)**

**Lecture Title: Computer Reliability (Cont.)**

**Software Engineering**

The field of software engineering grew out of a growing awareness of a “software crisis.” In the 1960s, computer architects had taken advantage of commercial integrated circuits to design much more powerful mainframe computers. These computers could execute much larger programs than their predecessors. Programmers responded by designing powerful new operating systems and applications. Unfortunately, their programming efforts were plagued by problems. The typical new software system was delivered behind schedule, cost more than expected, did not perform as specified, contained many bugs, and was too hard to modify. The informal, ad hoc methods of programming that worked fine for early software systems broke down when these systems reached a certain level of complexity. Software engineering is an engineering discipline focused on the production of software, as well as the development of tools, methodologies, and theories supporting software production. Software engineers follow a four-step process to develop a software product:

***Specification: defining the functions to be performed by the software***

***Development: producing the software that meets the specifications***

***Validation: testing the software***

***Evolution: modifying the software to meet the changing needs of the customer***

**Specification**

The process of specification focuses on determining the requirements of the system and the constraints under which it must operate. Software engineers communicate with the intended users of the system to determine what their needs are. They must decide if the software system is feasible given the budget and the schedule requirements of the customer. If a piece of software is going to replace an existing process, the software engineers study the current process to help them understand the functions the software must perform. The software engineers may develop prototypes of the user interface to confirm that the system will meet the user’s needs. The specification process results in a high-level statement of requirements and perhaps a mock-up of the user interface that the users can approve. The software engineers also produce a low-level requirements statement that provides the details needed by those who are going to actually implement the software system.

**Development**

During the development phase, the software engineers produce a working software system that matches the specifications. The first design is based on a high-level, abstract view of the system. The process of developing the high-level design reveals ambiguities, omissions, or outright errors in the specification. When these mistakes are discovered, the specification must be amended. Fixing mistakes is quicker and less expensive when the design is still at a higher, more abstract level. Gradually, the software engineers add levels of detail to the design. As this is done, the various components of the system become clear. Designers pay particular attention to ensure the interfaces between each component are clearly spelled out. They choose the algorithms to be performed and data structures to be manipulated. Since the emergence of software engineering as a discipline, a variety of structured design methodologies have been developed. These design methodologies result in the creation of large amounts of design documentation in the form of visual diagrams. Many organizations use computer-assisted software engineering (CASE) tools to support the process of developing and documenting an ever-more-detailed design. Another noteworthy improvement in software engineering methodologies is object oriented design. In a traditional design, the software system is viewed as a group of functions manipulating a set of shared data structures. In an object-oriented design, the software system is seen as a group of objects passing each other messages. Each object has its own state and manipulates its own data based on the messages it receives. Object-oriented systems have several advantages over systems constructed in a more traditional way:

1. Because each object is associated with a particular component of the system, object oriented designs can be easier to understand. More easily understood designs can save time during the programming, testing, and maintenance phases of a software project.
2. Because each object hides its state and private data from other objects, other objects cannot accidentally modify its data items.
3. Because objects are independent of each other, it is much easier to reuse components of an object-oriented system. A single object definition created for one software system can be copied and inserted into a new software system without bringing along other unnecessary objects.

When the design has reached a great enough level of detail, software engineers write the actual computer programs implementing the software system. Many different programming languages exist; each language has its strengths and weaknesses. Programmers usually implement object-oriented systems using an object-oriented programming language, such as C++, Java, or C#.

**Validation**

The purpose of validation (also called testing) is to ensure the software satisfies the specification and meets the needs of the user. In some companies, testing is an assignment given to newly hired software engineers, who soon move on to design work after proving their worth. However, good testing requires a great deal of technical skill, and some organizations promote testing as a career path. Testing software is much harder than testing other engineered artifacts, such as bridges.

We know how to construct scale models that we can use to validate our designs. To determine how much weight a model bridge can carry, we can test its response to various loads. The stresses and strains on the members and the deflection of the span change gradually as we add weight, allowing us to experiment with a manageable number of different loading scenarios. Engineers can extrapolate from the data they collect to generate predictions regarding the capabilities of a full-scale bridge. By increasing the size of various components, they can add a substantial margin of error to ensure the completed bridge will not fail.

A computer program is not at all like a bridge. Testing a program with a small problem can reveal the existence of bugs, but it cannot prove that the program will work when it is fed a much larger problem. The response of a computer program to nearly identical datasets may not be continuous. Instead, programs that appear to be working just fine may fail when only a single parameter is changed by a small amount. Yet programmers cannot exhaustively test programs. Even small programs have a virtually infinite number of different inputs. Since exhaustive testing is impossible, programs can never be completely tested. Software testers strive to put together suites of test cases that exercise all the capabilities of the component or system being validated.

To reduce the complexity of validating a large software system, testing is usually performed in stages. In the first stage of testing, each individual module of the system is tested independently. It is easier to isolate and fix the causes of errors when the number of lines of code is relatively small. After each module has been debugged, modules are combined into larger subsystems for testing. Eventually, all the subsystems are combined in the complete system. When an error is detected and a bug is fixed in a particular module, all the test cases related to the module should be repeated to see if the change that fixed one bug accidentally introduced another bug.

**Evolution**

Successful software systems evolve over time to meet the changing needs of their users. The evolution of a software system resembles the creation of a software system in many ways. Software engineers must understand the needs of the users, assess the strengths and weaknesses of the current system, and design modifications to the software. The same CASE tools used to create a new software system can aid in its evolution. Many of the datasets developed for the original system can be reused when validating the updated system.

Software Quality Is Improving There is evidence that the field of software engineering is becoming more mature (Figure 8.10). The Standish Group [56] regularly tracks thousands of IT projects. As recently as 1994, about one-third of all software projects were canceled before completion. About one-half of the projects were completed but had time and/or cost overruns, which were often quite large. Only about one-sixth of the projects were completed on time and on budget, and even in these cases the completed systems often had fewer features than originally planned. Another survey by the Standish Group in 2009 showed that the probability of a software project being completed on time and on budget had doubled, to about one in three. Only about one-quarter of the software projects surveyed were canceled. Slightly less than half of the projects were late and/or over budget, but the time and cost overruns were not as large as in the first survey. Overall, the ability of companies to produce software on time and on budget improved over this 15-year period.

Still, with only about one in three software projects being completed on time and on budget, the industry has a long way to go. Rapid change is a fact of life in the software industry. In order to stay competitive, companies must release products quickly. Many organizations feel a tension between meeting tight deadlines and strictly following software engineering methodologies.

**Software Warranties and Vendor Liability**

As mentioned earlier, Leveson and Turner state that “there is always another software bug”. If perfect software is impossible, what should the rights of consumers be to get compensation when programs malfunction? In this section we survey the software warranties offered by some software manufacturers, how these warranties have held up in court, and the variety of ways software vendors may be held liable for software defects.

**Shrink-wrap Warranties**

Consumer software is often called shrink-wrap software because of the plastic wrap surrounding the box containing the software and manuals. In the early years of personal computers, consumer software manufacturers provided no warranty for their products. Purchasers had to accept shrink-wrap software “as is.” Today many shrink-wrap software manufacturers provide a replacement or money-back guarantee if the program fails.

**Are Software Warranties Enforceable?**

How can software manufacturers get away with disclaiming any warranties on what they have sold? It’s not clear that they can. If the software is mass marketed or if it is included in a sale of hardware, it is likely to be considered a good by a court of law. The damages and warranty provisions of the Uniform Commercial Code (UCC) often apply to the sale of goods, despite what the warranties may say. An early court case, Step-Saver Data Systems v. Wyse Technology and The Software Link, seemed to affirm the notion that software manufacturers could be held responsible for defective programs, despite what they put in their warranties. However, two later cases seemed to indicate the opposite. In ProCD v. Zeidenberg, the court ruled that the customer was bound to the license agreement, even if the license agreement does not appear on the outside of the shrink-wrap box. Mortenson v. Timberline Software showed that a warranty disclaiming the manufacturer’s liability could hold up in court.

STEP-SAVER DATA SYSTEMS v. WYSE TECHNOLOGY AND THE SOFTWARE LINK Step-Saver Data Systems Inc. sold time-sharing computer systems consisting of an IBM PC AT server, Wyse terminals, and an operating system provided by the Software Link Inc. (TSL). In 1986–1987, Step-Saver purchased and resold 142 copies of the Multilink Advanced operating system provided by TSL.

To purchase the software, Step-Saver called TSL and placed an order, then followed up with a purchase order. According to Step-Saver, the TSL phone sales representatives said that Multilink was compatible with most DOS applications. The box containing the Multilink software included a licensing agreement in which TSL disclaimed all express and implied warranties.

Step-Saver’s time-sharing systems did not work properly, and the combined efforts of Step-Saver, Wyse, and TSL could not fix the problems. Step-Saver was sued by twelve of its customers. In turn, Step-Saver sued Wyse Technology and TSL.

The Third Circuit of the US Court of Appeals ruled in favor of Step-Saver. It based its argument on Article 2 of the UCC. The court held that the original contract between Step-Saver and TSL consisted of the purchase order, the invoice, and the oral statements made by TSL representatives on the telephone. The license agreement had additional terms that would have materially altered the contract. However, Step-Saver never agreed to these terms.

The court wrote, “In the absence of a party’s express assent to the additional or different terms of the writing, section 2-207 [of the UCC] provides a default rule that the parties intended, as the terms of their agreement, those terms to which both parties have agreed along with any terms implied by the provision of the UCC.” The court noted that the president of Step-Saver had objected to the terms of the licensing agreement. He had refused to sign a document formalizing the licensing agreement. Even after this, TSL had continued to sell to Step-Saver, implying that TSL wanted the business even if the contract did not include the language in the licensing agreement. That is why the court ruled that the purchase order, the invoice, and the oral statements constituted the contract, not the license agreement.

PROCD INC. v. ZEIDENBERG

ProCD invested more than $10 million to construct a computer database containing information from more than 3,000 telephone directories. ProCD also developed a proprietary technology to compress and encrypt the data. It created an application program enabling users to search the database for records matching criteria they specified. ProCD targeted its product, called SelectPhone, to two different markets: companies interested in generating mailing lists and individuals interested in finding the phone numbers or addresses of particular people they wanted to call or write. Consumers who wanted SelectPhone for personal use could purchase it for $150; companies paid much more for the right to put the package to commercial use. ProCD included in the consumer version of SelectPhone a license prohibiting the commercial use of the database and program. In addition, the license terms were displayed on the user’s computer monitor every time the program was executed.

Matthew Zeidenberg purchased the consumer version of SelectPhone in 1994. He formed a company called Silken Mountain Web Services Inc., which resold the information in the SelectPhone database. The price it charged was substantially less than the commercial price of SelectPhone. ProCD sued Matthew Zeidenberg for violating the licensing agreement.

At the trial, the defense argued that Zeidenberg could not be held to the terms of the licensing agreement, since they were not printed on the outside of the box containing the software. The US Court of Appeals for the Seventh Circuit ruled in favor of ProCD. Judge Frank Easterbrook wrote, “Shrinkwrap licenses are enforceable unless their terms are objectionable on grounds applicable to contracts in general (for example, if they violate a rule of positive law, or if they are unconscionable)”.

MORTENSON v. TIMBERLINE SOFTWARE M.

Mortenson Company was a national construction contractor with a regional office in Bellevue, Washington. Timberline Software Inc. produced software for the construction industry. Mortenson had used software from Timberline for several years. In July 1993, Mortenson purchased eight copies of a bidding package called Precision Bid Analysis.

In December 1993, Mortenson used Precision Bid Analysis to prepare a bid for the Harborview Medical Center in Seattle. On the day the bid was due, the software malfunctioned. It printed the message “Abort: Cannot find alternate” 19 times. Mortenson continued to use the software and submitted the bid the software produced. After the firm won the contract, Mortenson discovered that its bid was $1.95 million too low.

Mortenson sued Timberline for breach of express and implied warranties. It turns out Timberline had been aware of the bug uncovered by Mortenson since May 1993. Timberline had fixed the bug and already sent a newer version of the program to some of its other customers who had encountered it. It had not sent the improved program to Mortenson. Nevertheless, Timberline argued that the lawsuit be summarily dismissed because the licensing agreement limited the consequential damages that Mortsenson could recover from Timberline. The King County Superior Court ruled in favor of Timberline. The ruling was upheld by the Washington Court of Appeals and the Supreme Court of the State of Washington.

***Reference***

***Lecture 27 slides: Software Engineering***

***Gao, Y. (2012). Ethics for the Information Age by Michael J. Quinn. World Libraries, 20(1).***