Software Engineering 2014

Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering

A Volume of the Computing Curricula Series

23 February 2015

.

Joint Task Force on Computing Curricula
IEEE Computer Society
Association for Computing Machinery

Preface

This document was developed through an effort originally commissioned by the ACM Education Board and the IEEE-Computer Society Educational Activities Board to create curriculum recommendations in several computing disciplines: computer science, computer engineering, software engineering and information systems. Other professional societies have joined in a number of the individual projects. Such was the case for the SE2004 (Software Engineering 2004) project, which included participation by representatives from the Australian Computer Society, the British Computer Society, and the Information Processing Society of Japan.

SE2004 Development Process

The SE2004 project was driven by a Steering Committee appointed by the sponsoring societies. The development process began with the appointment of the Steering Committee co-chairs and a number of the other participants in the fall of 2001. More committee members, including representatives from the other societies were added in the first half of 2002. The following are the members of the SE2004 Steering Committee:

Co-Chairs

Rich LeBlanc, ACM, Georgia Institute of Technology, U.S.
Ann Sobel, IEEE-CS, Miami University, U.S.

Knowledge Area Chair

Ann Sobel, Miami University, U.S.

Pedagogy Focus Group Co-Chairs

Mordechai Ben-Menachem, Ben-Gurion University, Israel
Timothy C. Lethbridge, University of Ottawa, Canada

Co-Editors

Jorge L. Díaz-Herrera, Rochester Institute of Technology, U.S.
Thomas B. Hilburn, Embry-Riddle Aeronautical University, U.S.

Organizational Representatives

ACM: Andrew McGettrick, University of Strathclyde, U.K.
ACM SIGSOFT: Joanne M. Atlee, University of Waterloo, Canada
ACM Two-Year College Education: Elizabeth K. Hawthorne, Union County College, U.S.
Australian Computer Society: John Leaney, University of Technology Sydney, Australia
British Computer Society: David Budgen, Keele University, U.K.
Information Processing Society of Japan: Yoshihiro Matsumoto, Musashi Institute of Technology, Japan
IEEE-CS Technical Committee on Software Engineering: J. Barrie Thompson, University of Sunderland, U.K.

SE2014 Revision Process

This updated version of the curriculum guidelines was created by a joint effort of the ACM and the IEEE-Computer Society:

IEEE CS Delegation
Mark Ardis, *Chair* (Stevens Institute)
Greg Hislop (Drexel University)
Mark Sebern (Milwaukee School of Engineering)

ACM Delegation
David Budgen (University of Durham)
Jeff Offutt (George Mason University)
Willem Visser (University of Stellenbosch)

Acknowledgements

The National Science Foundation, the Association of Computing Machinery, and the IEEE Computer Society have supported the development of this document and its predecessor, SE2004.

We thank the many reviewers and technical experts who provided feedback and advice for this update, starting with those who contributed suggestions for improvements to SE2004. An online anonymous survey on that subject yielded 477 responses from software engineering educators and practitioners in 42 different countries. In addition to those participants we thank Jo Atlee and Renee McCauley for their assistance in assessing and planning the revision effort that followed.

We thank the many participants at workshops and feedback sessions at SIGCSE 2013, CSEE&T 2013 and ICSE 2013. Their comments were particularly helpful in confirming the revisions that had already been made, and in suggesting additional improvements that we have incorporated into the final document.

We are especially thankful for the extensive comments offered by expert reviewers Dennis Frailey, Tom Hilburn, Rich LeBlanc, Andrew McGettrick and Nancy Mead. We thank Steve Chenoweth from Rose-Hulman Institute of Technology, Sarah Lee from Mississippi State University, and Frank Tsui of Southern Polytechnic State University for providing examples of software engineering courses and curricula. We hope that this collection of examples will grow larger on the accompanying website provided by the ACM Education Board. We are grateful for the expert assistance provided by the IEEE-CS Technical Publications staff in producing a more coherent and consistent document.

Finally, we thank the members of the ACM and IEEE-CS education boards who supported this effort, especially Andrew McGettrick from the Association for Computing Machinery and Tom Hilburn from the IEEE Computer Society who initiated the project.

Table of Contents

Chapter 1. Introduction

1.1 Purpose of this Volume

The primary purpose of this volume is to provide guidance to academic institutions and accreditation agencies about what should constitute an undergraduate software engineering education. These recommendations were originally developed by a broad, international group of volunteer participants. Software engineering curriculum recommendations are of particular relevance because the number of new software engineering degree programs continues to grow steadily and accreditation processes for such programs have been established in a number of countries.

The recommendations included in this volume are based on a high-level set of characteristics recommended for software engineering graduates, which are presented in Chapter 3. Flowing from these outcomes are the two main contributions of this document:

- The Software Engineering Education Knowledge (SEEK): what every SE graduate must know.
- Curriculum: ways this knowledge and the skills fundamental to software engineering can be taught in various contexts.

1.2 Where This Volume Fits in the Computing Curriculum Context

In 1998, the Association for Computing Machinery (ACM) and IEEE Computer Society (IEEE CS) convened a joint curriculum task force called the *Computing Curricula 2001* (CC 2001). In its original charge, the CC 2001 Task Force was asked to develop a set of curricular guidelines that would "match the latest developments of computing technologies in the past decade and endure through the next decade." The members of this task force recognized early in the process that they, as a group primarily consisting of computer scientists, were ill-equipped to produce guidelines that would cover computing technologies in their entirety. Over the past 50 years, computing has become an extremely broad designation that extends well beyond the boundaries of computer science to encompass such independent disciplines as computer engineering, software engineering, information systems, and many others. Given the breadth of that domain, the curriculum task force concluded that no group representing a single specialty could hope to do justice to computing as a whole. At the same time, feedback received on an initial draft made it clear that the computing education community strongly favored a report that did take into account the breadth of the discipline.

Their solution to this challenge was to continue work on the development of a volume of computer science curriculum recommendations, published in 2001 as the *CC 2001 Computer Science* volume (CCCS volume)[IEEE 2001b]. In addition, the task force recommended to the sponsoring organizations that the project be broadened to include volumes of recommendations for the related disciplines previously listed as well as any

others that might be deemed appropriate by the computing education community. In this context, this document containing curriculum guidelines for software engineering was initially developed and continues to evolve.

1.3 Development Process of the SE 2014 Volume

The first set of guidelines for software engineering curricula was published in 2004 [IEEE 2004]. In 2010, a task force was appointed by the ACM and IEEE CS to determine whether updates were needed and, if so, how much effort would be required to complete them. The task force reached out to academia, industry, and government through workshops at technical conferences and an online survey. It was determined that a small team could make the needed updates during the following year.

Once the revision team was formed, its members identified sections of the original guidelines that needed updating and started to make revisions. During this process, they continued to reach out to stakeholders through presentations and workshops at technical conferences. At one such workshop, held at the 2013 Conference on Software Engineering Education and Training, they presented an initial draft of proposed revisions to the SEEK and other areas of the curriculum guidelines. Based on the positive feedback obtained at that workshop, they continued their revisions and prepared a draft for public review in the fall of 2013. Additional revisions were made in response to feedback from that review.

1.4 Changes from the Previous Version

This new version of the curriculum guidelines shares much of the original structure of the previous version, SE2004. Chapter 2 was rewritten to reflect an improved understanding of the discipline of software engineering as it has evolved over the last ten years. The guiding principles in Chapter 3 were reordered and given tags so that they could be more easily referred to and applied. The overall structure of the SEEK in Chapter 4 remains the same, but modifications were made to reflect changes in the field. In particular, this version recognizes the emergence of alternative lifecycle process models, including those with the increased agility required in many contemporary application domains. The new version also increases the visibility of software requirements and security, as those topics have become of increasing interest and concern.

Some of the advice in later chapters of the guidelines has been simplified to remove generic instructional and curricular advice. Rather, specific topics relevant to the teaching of software engineering have been retained and updated to include recent advances in teaching technologies, such as massive open online courses (MOOCs).

Finally, a collection of example courses and curricula has been included as appendices. When SE2004 was written there were very few undergraduate software engineering programs, so examples of courses were largely speculative. In the last ten years, a significant number of programs have been initiated, providing a rich source of successful courses and curricula to share.

1.5 Structure of the Volume

Chapter 2 discusses the nature and evolution of software engineering as a discipline, identifies some of its key elements, and explains how these elements have influenced the recommendations in this document. Chapter 3 presents the guiding principles, adapted from those originally articulated by the CC 2001 Task Force, that have supported the development of these curriculum guidelines. Chapter 4 presents the body of Software Engineering Education Knowledge (SEEK) that underlies the curriculum guidelines (Chapter 5) and educational program designs (Chapter 6). Chapter 7 discusses adaptation of the curriculum guidelines to alternative environments. Chapter 8 addresses various curriculum implementation challenges and considers assessment approaches.

Following a practice adopted in the most recent version of the curriculum guidelines for undergraduate computer science programs [CS2013], the appendices of this report contain example curricula and courses from existing undergraduate software engineering programs.

Chapter 2: The Software Engineering Discipline

This chapter discusses the nature of software engineering and some of the history and background that is relevant to the development of software engineering curriculum guidance. The purpose of the chapter is to provide context and rationale for the curriculum materials in subsequent chapters.

2.1 Defining Software Engineering

Since the dawn of electronic computing in the 1940s, computing systems and their applications have evolved at a staggering rate. Software plays a central and underpinning role in almost all aspects of daily life: communications, government, manufacturing, banking and finance, education, transportation, entertainment, medicine, agriculture, and law. The number, size, and application domains of computer programs have grown dramatically; as a result, huge sums are being spent on software development [OECD 2010]. Most people's lives and livelihoods depend on this development's effectiveness. Software products help us to be more efficient and productive. They provide information, make us more effective problem solvers, and provide us with safer, more flexible, and less confining work, entertainment, and recreation environments.

Despite these successes, this period has witnessed serious problems in terms of the development costs, timeliness, and quality of many software products. There are many reasons for these problems:

- Software products are among the most complex manmade systems, and by its very nature, software has intrinsic, essential properties (for example, complexity, invisibility, and changeability) that are not easily addressed [Brooks 1987].
- Programming techniques and processes that work effectively when used by an individual or small team to develop modest-sized programs do not scale well to the development of large, complex systems. (Complexity can arise with just a few hundred lines of code, and large systems can run to millions of lines of code, requiring years of work by hundreds of software developers.)
- The pace of change in computer and software technology drives the demand for new and evolved software products. This situation has created customer expectations and competitive forces that strain our ability to produce quality software within acceptable development schedules.
- The availability of qualified software engineers has not kept pace with the demand from industry, so that systems are designed and built by people with insufficient educational background or experience.

The term "software engineering" [Naur 1969] has now become widely used in industry, government, and academia. Hundreds of thousands of computing professionals go by the title "software engineer"; numerous publications, groups and organizations, and professional conferences use the term software engineering in their names; and many

educational courses and programs on software engineering are available. Unfortunately, as with the term "engineer" itself, the term software engineer is not always used to mean "a software engineering professional," although that is the meaning that is assumed throughout this document.

Over this same period, although the software engineering discipline has evolved, the context has changed as well. In the 1960s, a software product was usually created as a single, monolithic entity, executed on a computer with the support of a fairly basic operating system. Such a product had external operations that were mainly confined to basic file input/output. In contrast, a software system developed in today may well reuse major components of other systems, execute on multiple machines and platforms, and interact with other, globally distributed systems [da Silva 2012].

Thus, although the current generation of software engineers undertake many of the same activities as their predecessors, they are likely to do so in a more complex environment. In addition, the consequences of any changes made to a system in the 1960s were likely to be localized, whereas now there may be truly global effects.

Over the years, ideas about what exactly software engineering is have also evolved. Nevertheless, a common thread exists that states (or strongly implies) that software engineering is more than just programming; it includes attention to details such as quality, schedule, and economic goals. Hence, a professional software developer needs both knowledge of such principles and experience with applying them.

The fact that the literature contains many different definitions of software engineering implies that a concise and complete definition of software engineering is difficult to formulate. This is largely because the interpretation of any definition requires an understanding, not only of the unique characteristics of software, but also of how "engineering" concepts must be adapted to address those characteristics.

The IEEE's 2010 definition states that software engineering is

> The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software. [IEEE 2010]

The nature of software, however, complicates the interpretation of these words [Brooks 1987]. To address this ambiguity, it is helpful to identify some key characteristics of software and the associated challenges that they create for any form of "engineering" process.

- Software is *abstract* and *invisible*. These characteristics present challenges for managing software development because of the problems they create for important engineering concepts such as quantification and measurement. They also complicate efforts to describe and organize software in ways that will facilitate knowledge exchange during the processes of its design, implementation, and maintenance.

- Software has both *static* and *dynamic* properties. This duality provides a further challenge for description and measurement. It also makes it difficult to predict the effects arising from any changes made to a software product.
- Software is *intrinsically complex* in terms of its organization. Even a small software unit may possess many different execution paths, and there may be a large and complex set of relationships among its elements. This in turn presents challenges for verification and validation, documentation, and maintenance.
- No universal measures of *quality* exist for assessing a software product [Hughes 2000]. An engineering process should lead to products that are of "good" quality, but the relative importance of different quality measures will vary with the role of the product and differ for each stakeholder (producer, customer, or user).
- The *manufacturing* cycle for software products is not a significant element in software development, and it mainly involves the needs of distribution mechanisms. Software development is essentially a process that involves a progression through many layers of design abstraction and, hence, is unlike any conventional engineering processes, such as those that occur within mechanical and civil engineering.
- Software does *not wear out*. The maintenance activities associated with a software product are really part of an evolutionary design process.

Essentially therefore, software engineering practices are largely concerned with *managing* relevant processes and with *design* activities, and these can appear in a range of guises. Most of the activities involved in software development and evolution tend to use team-based processes that embody some form of design element, spanning from an initial choice of a high-level architecture through the choices of test and evaluation strategies. Each of these adds yet another layer of complication: teams must be organized with regard to aspects such as communication, coordination, and management and design activities are nondeterministic (or "wicked") processes that lead to solutions that are rarely right or wrong [Rittel & Webber 1984; Peters & Tripp 1976]. Finally, there are also many different measures of quality that can be employed when assessing the different choices involved.

2.2 The Evolution of Software Engineering

Software engineering concepts in the early years were largely dominated by the idea of structure, both in terms of the product and the processes that were recommended as the means of creating that product. This thinking was largely characterized by the idea of the *waterfall model*, with each phase (such as requirements specification, design, implementation, testing, and documentation) needing to be completed before the next one could begin. However, early on it became clear that, while this might be an appropriate description for the way that some types of software system were produced, particularly those with a well-understood and specified role and purpose, such models were too rigid, especially when they were used in a prescriptive manner [Gladden 1982; McCracken & Jackson 1982].

The evolution of more flexible ways to organize software development, while retaining an appropriate degree of discipline in the process, went through various phases. One of

the early ideas was *prototyping* [Floyd 1984]. A prototype could be used to explore both the problem and design spaces (the exploratory and experimental roles), as in other branches of engineering. In the case of software engineering, the prototype could also evolve to become the final product. The risk of the latter becoming a "code and fix" approach was also recognized and addressed through such refinements as the *spiral model* [Boehm 1988]. However, these approaches were essentially incremental modifications to the waterfall approach.

The widening spectrum of software applications, especially in business and commerce, and the emergence of the Internet encouraged the adoption of more flexible iterative and incremental forms of development during the 1990s. This was characterized by terms such as rapid application development (RAD) and the emergence of software tools intended to assist such processes. Associated with this was a growing recognition that the "customer" needed to be involved in the development, rather than being simply a provider of a set of requirements that were fixed at the start of a project.

The subsequent emergence of the *agile concept* and the Agile Manifesto offered a more revolutionary view of development [Boehm & Turner 2003; Beck 2004; Schwaber 2004, Holcombe 2008]. As its proponents have rightly emphasized, agile thinking does not discard earlier ideas and concepts; it adjusts the emphasis given to different aspects of the development process allows for greater adaptability of form. Some process artifacts, such as extensive documentation, are deemphasized, and the people in the process (developers, customers, and other stakeholders) and their interactions are typically given greater priority.

Not only the processes have changed. The increased use, scope, and availability of *open source software* (OSS) has created both economic and social changes in expectation and motivation for software professionals [Tucker et al. 2011]. It has also created greater opportunities for employing *component-based approaches* in the development of software systems.

Consequently, although software engineers of today continue to perform many of the same activities as in the 1960s, they do so in a very different context. Not only are the programming languages and tools that they employ for these activities different, and generally much more powerful, but these activities are also likely to be undertaken within a much wider variety of organizational models. For example, software for safety-critical systems, where strong quality control is essential, is still likely to be produced using a formally controlled waterfall-style approach, whereas software for Web-based applications that needs to reach the marketplace quickly may use lightweight agile forms, allowing quick responses to changing conditions and new opportunities. In addition, a software development team might not even be colocated, with *global software development* (GSD) practices also opening up the possibility of working across different time zones [Smite et al. 2010].

The guidelines provided in this document do not advocate any one developmental context or process, not the least because knowledge of different processes is part of a software

engineer's education. As much as possible, the guidelines address the activities involved in developing software without making any assumptions about how they are organized within the overall development process.

Another aspect of the discipline that has evolved with its growing maturity is the nature of software engineering knowledge itself. Early approaches relied on expert interpretation of experience ("structured" approaches) as well as the use of models based on mathematical formalism ("formal" approaches). More recently, widespread practical experience and empirical studies have contributed to a better understanding of how and when these models and ideas work, how they need to be adapted to varying situations and conditions, and the importance of human interactions when defining and developing software systems [Glass et al. 2004; Pfleeger 2005].

Hence, software engineering knowledge now takes many forms and is codified in many different ways. It is usually expressed in terms of a mix of models (formal and informal), expert experience, and empirical assessments, as best suited to the aspects of interest. This increased diversity is reflected in the set of reference disciplines described in the next section.

2.3 The Reference Disciplines for Software Engineering

As discussed earlier, producing software in a systematic, controlled, and efficient manner and being able to do so for a range of applications and contexts requires an extensive range of "tools" (both conceptual and physical) together with the necessary understanding of how best to deploy them. The underpinnings for this are drawn from a range of disciplines, and some significant contributions and influences are therefore discussed in this section.

Software Engineering as a Computing Discipline

In the early days of computing, computer scientists produced software, and computer engineers built the hardware to host its execution. As the size, complexity, role, and critical importance of software artifacts grew so did the need to ensure that they performed as intended. By the early 1970s, it was apparent that proper software development practices required more than just the underlying principles of computer science; they needed both the analytical and descriptive tools developed within computer science and the rigor that the engineering disciplines bring to the reliability and trustworthiness of the artifacts they engineer.

Drawing on computing as one of its foundations, software engineering seeks to develop and use systematic models and reliable techniques to produce high-quality software. These concerns extend from theory and principles to the development practices that are most visible to those outside the discipline. Although it is unlikely that every software engineer will have deep expertise in all aspects of computing, a general understanding of each aspect's relevance and some expertise in particular aspects are a necessity. The definition of the body of the SEEK described in Chapter 4 reflects the reliance of

software engineering on computer science, with the largest SEEK component being computing essentials.

Software Engineering as an Engineering Discipline

The study and practice of software engineering is influenced both by its roots in computer science and its emergence as an engineering discipline. A significant amount of current software engineering research is conducted within the context of computer science and computing departments or colleges. Similarly, software engineering degree programs are being developed by such academic units as well as in engineering colleges. Thus, software engineering maintains a stronger connection to its underlying discipline (computer science) than may be the case for some other engineering fields. In the process of constructing this volume, particular attention has been paid to incorporating engineering practices into software development so as to distinguish software engineering curricula from those appropriate to computer science degree programs. To prepare for the more detailed development of these ideas, this section considers how the engineering methodology applies to software development.

Some critical characteristics common to every other engineering discipline are equally applicable to software engineering. Thus, they have influenced both the development of software engineering and the contents of this volume.

[1] Whereas scientists observe and study existing behaviors and then develop models to describe them, engineers use such models as a starting point for designing and developing technologies that enable new forms of behavior.

[2] Engineers proceed by making a series of decisions, carefully evaluating options, and choosing an approach at each decision point that is appropriate for the current task in the current context. Appropriateness can be judged by trade-off analysis, which balances costs against benefits.

[3] Engineers measure things, and when appropriate, work quantitatively. They calibrate and validate their measurements, and they use approximations based on experience and empirical data.

[4] Engineers emphasize the use of a disciplined process when creating and implementing designs and can operate effectively as part of a team in doing so.

[5] Engineers can have multiple roles: research, development, design, production, testing, construction, operations, and management in addition to others such as sales, consulting, and teaching.

[6] Engineers use tools to apply processes systematically. Therefore, the choice and use of appropriate tools is a key aspect of engineering.

[7] Engineers, via their professional societies, advance by the development and validation of principles, standards, and best practices.

[8] Engineers reuse designs and design artifacts.

Although strong similarities exist between software engineering and more traditional engineering, there are also some differences (not necessarily to the detriment of software engineering):

- Software engineering's foundations are primarily in computer science, not natural sciences.
- Software engineering models make more use of discrete than continuous mathematics.
- The concentration is on abstract/logical entities instead of concrete/physical artifacts.
- There is no manufacturing phase in the traditional sense.
- Software maintenance primarily refers to continued development, or evolution, and not to conventional wear and tear.
- Software engineering is not always viewed as a "professional" activity. One concern for these curriculum guidelines is to help with the evolution of software engineering toward a more "professional" status.

In using the term engineer and engineering extensively, this document is about the design, development, and implementation of undergraduate software engineering curricula.

Mathematics and Statistics

Software engineering makes little direct use of traditional continuous mathematics, although such knowledge may be necessary when developing software for some application domains as well as when learning statistics. Like computer science, software engineering makes use of discrete mathematical formalisms and concepts where necessary, such as when modeling the interactions and potential inconsistencies among different requirements and design solutions, modeling artifacts for test design, and modeling behavior for security analysis.

Statistics also have a role in software engineering. Activities such as cost modeling and planning require an understanding of probability, and interpretation of the growing body of empirical knowledge similarly needs familiarity with issues such as significance and statistical power. In addition, the interactions of a software artifact with other system elements often leads to behavior that is nondeterministic and, hence, best described using statistical models. Because these are all applications of statistics and probability, a calculus-based treatment is not necessarily required.

Psychology and the Social Sciences

Interpersonal relations play a central role in many software engineering activities. Although the reality of this, and the significance of the ways that groups and teams function, was recognized early on in the development of the subject [Weinberg 68], it tended to be downplayed by the plan-driven approaches to software development that were structured around the waterfall model. Consideration of human factors was therefore largely limited to aspects such as human-computer interaction and project management, which remain important today.

One of the key contributions from contemporary iterative and agile thinking has been a greater emphasis on the people in the software development process, including customers, users, and other stakeholders as well as software engineering professionals, both in terms of their roles and the interactions between them [Beecham et al. 2008; Leffingwell 2011]. This has been accompanied by a recognition that the interactions between users and systems must be a fundamental element of design thinking and that this should focus on exploiting different viewpoints rather than constraining them [Rogers et al. 2011].

Although, for the purposes of curriculum design, these are not subject areas needing deep study, software engineers must be aware of the effects that human factors can have across many of the discipline's activities. As such, these crosscutting concerns inform the presentation and discussion of many of the topics that make up the SEEK.

Management Science

All software development projects need to be managed in some way, even if only one person is involved. Planning, estimation, and version control (release management) are examples of management activities that are necessary for any project, regardless of its size or chosen process model. Similarly, team management and the effects of factors such as individual and team motivation are important issues for nearly every project.

The absence of any significant manufacturing stage for software changes the nature of project management in software engineering, and agile practices may require different management tasks than plan-driven approaches. Nevertheless, software projects must still be managed, even if some adaptation of management science concepts and models is required.

2.4 Professional Practice

A key objective of any engineering program is to provide graduates with the tools necessary to begin professional engineering practice. As Chapter 3 indicates, an important guiding principle for this document is that software engineering education should include student experiences with the professional practice of software engineering. Subsequent chapters discuss the content and nature of such experiences, while this section provides rationale and background for the inclusion of professional practice elements in a software engineering curriculum.

Rationale

Professionals have special obligations requiring them to apply specialist knowledge on behalf of members of society who do not have such knowledge. All the characteristics of engineering discussed in Section 2.3 relate, directly or indirectly, to the professional practice of engineering. Employers of engineering program graduates often speak to these same needs [Denning 1992]. Each year, the National Association of Colleges and Employers conducts a survey to determine what qualities employers consider most important in applicants seeking employment. In 2013, employers were asked to rate the

importance of candidate qualities and skills on a five-point scale, with five being "extremely important" and one being "not important." Communication skills (4.63 average), ability to work in a team (4.6), problem-solving skills (4.51), planning and organizational skills (4.46), ability to obtain and process information (4.43), and ability to analyze quantitative data (4.3) were the most desired characteristics [NACE 2013].

Graduates of software engineering programs need to arrive in the workplace equipped to meet the challenges of society's critical dependence on software and to help evolve the standards and practices of the software engineering discipline. Like other engineering professionals, when it is appropriate and feasible, software engineers should seek to base decisions on quantitative data, but they must also be able to function effectively in an environment of ambiguity and avoid the limitations of oversimplified or unverified modeling.

Software Engineering Code of Ethics and Professional Practices

Software engineering as a profession has obligations to society. The products produced by software engineers affect the lives and livelihoods of their clients and the product users. Hence, software engineers need to act ethically and professionally. The preamble to the joint ACM/IEEE *Software Engineering Code of Ethics and Professional Practice* [ACM 1998] states,

> Because of their roles in developing software systems, software engineers have significant opportunities to do good or cause harm, to enable others to do good or cause harm, or to influence others to do good or cause harm. To ensure, as much as possible, that their efforts will be used for good, software engineers must commit themselves to making software engineering a beneficial and respected profession. In accordance with that commitment, software engineers shall adhere to the following Code of Ethics and Professional Practice.

To help ensure ethical and professional behavior, software engineering educators have an obligation not only to make their students familiar with this code, but also to find ways for students to engage in discussion and activities that illustrate and illuminate the code's eight principles, including common dilemmas facing professional engineers in typical employment situations.

Curriculum Support for Professional Practice

A curriculum can have an important, direct effect on some professional practice factors (such as teamwork, communication, and analytic skills), while others (such as a strong work ethic and self-confidence) are subject to the more subtle influence of a college education on an individual's character, personality, and maturity. In this volume, Chapter 4 identifies elements of professional practice that should be part of any curriculum and expected student outcomes. Chapters 5 and 6 contain guidance and ideas for incorporating material about professional practice into a software engineering curriculum.

Many elements, some outside the classroom, can significantly affect a student's preparation for professional practice. Examples include involvement in the core curriculum by faculty with professional experience; student work experience as an intern or in a cooperative education program; and extracurricular activities, such as technical presentations, field trips, visits to industry, and activities sponsored by student professional organizations.

2.5 Foundations and Related Work

The task of determining the curriculum guideline's scope and content has drawn upon a range of sources. Where appropriate, sources are cited in the relevant chapters. These sources include prior initiatives that have sought to codify knowledge in software engineering and related areas:

- The original *2004 Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering* was developed by the IEEE CS and ACM. This in turn was an element of the "Computing Curricula 2005," which developed guidelines for programs in computer engineering, computer science, information systems, information technology, and software engineering.
- The *Conference on Software Engineering Education & Training* (CSEE&T), originally initiated by the Software Engineering Institute in 1987, is now one of many software engineering conferences sponsored by the IEEE CS, ACM, and other professional bodies.
- The *Guide to the Software Engineering Body of Knowledge* (SWEBOK) was produced by the IEEE CS to identify a body of knowledge needed for the practice of software engineering [SWEBOK 2014], and it also undergoes periodic updating and revision (www.swebok.org). One of the objectives of the SWEBOK project was to "provide a foundation for curriculum development." To support this objective, SWEBOK includes a rating system for its knowledge topics based on Bloom's levels of educational objectives [Bloom 1956]. It should be noted, however, that SWEBOK is intended to cover the level of knowledge acquired after four years of practice and also intentionally does not address non-software-engineering knowledge that a software engineer must have.
- The development of ideas about the field of *systems engineering* is in turn dependent upon software engineering in many ways. Although there are no undergraduate curriculum guidelines for systems engineering, a *Graduate Reference Curriculum for Systems Engineering* (GRCSE) has been developed (www.bkcase.org/grcse).

Chapter 3: Guiding Principles

This chapter describes three key aspects underpinning the curriculum guidelines. The first is the desired outcomes for a student who has studied an undergraduate curriculum in software engineering. The second is a set of foundational ideas and beliefs about the nature and form of software engineering. The third concerns the goals for the curriculum guidelines. Together, these have helped to determine the choice and organization of the SE 2014 materials.

3.1 Expected Student Outcomes

As a basic step toward providing curriculum guidance, the following set of outcomes for an undergraduate curriculum has been identified. This is intended as a generic list that could be adapted to various software engineering program implementations. Although emphasis is placed on knowledge and skills related to software engineering, an undergraduate curriculum should of course enable a student to analyze and synthesize these elements as appropriate.

Graduates of an undergraduate SE program should be able to demonstrate the following qualities.

[**Professional Knowledge**] *Show mastery of software engineering knowledge and skills and of the professional standards necessary to begin practice as a software engineer*.

Students, through regular reinforcement and practice, need to gain confidence in their abilities as they progress through a software engineering program of study. In most instances, students acquire knowledge and skills through a staged approach in which they achieve different levels as each academic term progresses. In addition, graduates must have an understanding and appreciation of professional issues and standards related to ethics and professional conduct, economics, and societal needs.

[**Technical Knowledge**] *Demonstrate an understanding of and apply appropriate theories, models, and techniques that provide a basis for problem identification and analysis, software design, development, implementation, verification, and documentation.*

Software engineering employs concepts that are unique to the nature of software and its development and also draws others from a range of reference disciplines. Students should both be aware of these concepts and of their limitations, whether inherent or arising from their adaptation to software engineering. Students should be able to evaluate and reflect on the processes that they follow as well as upon the solutions that they produce.

[**Teamwork**] *Work both individually and as part of a team to develop and deliver quality software artifacts.*

Students need to perform tasks that involve working as an individual, but also experience many other tasks that entail working with a group. For group work, students should be informed about the nature of groups and of group activities and

roles as explicitly as possible. This must include an emphasis on the importance of such matters as a disciplined approach, adhering to deadlines, communication, and individual and team performance evaluations.

[**End-User Awareness**] *Demonstrate an understanding and appreciation of the importance of negotiation, effective work habits, leadership, and good communication with stakeholders in a typical software development environment.*

A program of study should include at least one major activity that involves producing a solution for a client. Software engineers must take the view that they have to produce software that is of genuine utility. Where possible, a program should incorporate a period of industrial experience as well as invited lectures from practicing software engineers and involvement in activities such as external software competitions. All this provides a richer experience and helps to create an environment that supports the development of high-quality software engineering graduates.

[**Design Solutions in Context**] *Design appropriate solutions in one or more application domains using software engineering approaches that integrate ethical, social, legal, and economic concerns.*

Throughout their study, students should be exposed to a variety of appropriate approaches to engineering design in the general sense and to examples of their use in developing software for different application domains. They must be able to understand the strengths and limitations of the available options and the implications of selecting specific approaches for a given situation. Their proposed design solutions must be developed within the context of ethical, social, legal, security, and economic concerns.

[**Perform Trade-Offs**] *Reconcile conflicting project objectives, finding acceptable compromises within the limitations of cost, time, knowledge, existing systems, and organizations.*

Students should engage in exercises that expose them to conflicting and changing requirements. There should be a strong real-world element present in such cases to ensure that the experience is realistic. Curriculum units should address these issues, with the aim of ensuring high-quality functional and nonfunctional requirements and a feasible software design.

[**Continuing Professional Development**] *Learn new models, techniques, and technologies as they emerge and appreciate the necessity of such continuing professional development.*

By the end of their program of study, students should show evidence of being self-motivated lifelong learners. Throughout a program of study, students should be encouraged to seek new knowledge and to appraise it for usefulness and relevance.

3.2 SE 2014 Principles

The following list of principles embraces both general computing principles as well as those that reflect the special nature of software engineering and that differentiate it from other computing disciplines.

[**Software Engineering in the Computing Spectrum**] *Computing is a broad field that extends well beyond the boundaries of any one computing discipline*.

SE 2014 concentrates on the knowledge and pedagogy associated with a software engineering curriculum. Where appropriate, it will share or overlap with material contained in other computing curriculum reports and will offer guidance on its incorporation into other disciplines.

[**Reference Disciplines**] *Software Engineering draws its foundations from a variety of disciplines*.

Undergraduate study of software engineering relies on many areas in computer science for its theoretical and conceptual foundations, but it also requires students to use concepts from other fields, such as mathematics, engineering, and project management. All software engineering students must learn to integrate theory and practice, recognize the importance of abstraction and modeling, appreciate the value of good design, and be able to acquire special domain knowledge beyond the computing discipline for the purposes of supporting software development in specific application domains.

[**Curriculum Evolution**] *The continuing evolution of software engineering knowledge, technology, applications, pedagogy, and practices together with the professional nature of software engineering require an ongoing review of the corresponding curriculum and an emphasis upon the importance of lifelong learning for graduates.*

To address the continuously evolving nature of software engineering, educational institutions must adopt explicit strategies for responding to change. This should include an ongoing review process that allows individual components of the curriculum recommendations to be updated on a recurring basis. Institutions, for example, must recognize the importance of remaining abreast of well-established progress in both technology and pedagogy, subject to the constraints of available resources. Software engineering education, moreover, must seek to prepare students for lifelong learning that will enable them to move beyond today's technology to meet the challenges of the future.

[**Curriculum Organization**] *SE 2014 must go beyond knowledge elements to offer significant guidance in terms of individual curriculum components*.

The SE 2014 curriculum models should assemble the knowledge elements into reasonable, easily implemented learning units. Articulating a set of well-defined curriculum models will make it easier for institutions to share pedagogical strategies and tools. It will also provide a framework for publishers who provide textbooks and other materials.

[**Software Engineering Core**] *SE 2014 must support the identification of the fundamental skills and knowledge that all software engineering graduates must possess.*

Where appropriate, SE 2014 must help define the common themes of the software engineering discipline and ensure that all undergraduate program recommendations include this material.

[**Incorporation of Software Engineering Knowledge**] *Guidance on software*

engineering curricula must be based on an appropriate definition of software engineering knowledge.

The description of this knowledge should be concise and appropriate for undergraduate education and should use the work of previous studies on the software engineering body of knowledge. From this description, a core set of required topics must be specified for all undergraduate software engineering degrees. The core should have broad acceptance by the software engineering education community. Coverage of the core will start with the introductory courses, extend throughout the curriculum, and be supplemented by additional courses that may vary by institution, degree program, or individual student.

3.3 SE 2014 Goals for the Guidelines

For both the original guidelines and these revisions, there have been a number of overarching goals that relate to the scope of the curriculum guidelines.

[**International Relevance**] *SE 2014 must strive to be international in scope.*

Although curricular requirements and structures may differ from country to country, SE 2014 must be useful to computing educators throughout the world. Where appropriate, every effort should be made to ensure that the curriculum recommendations are sensitive to national and cultural differences so that they will be widely applicable throughout the world. The involvement by national computing societies and volunteers from all countries should be actively sought and welcomed.

[**Range of Perspectives**] *The development of SE 2014 must be broadly based*.

To be successful, the process of creating software engineering education recommendations must consider the many perspectives represented by software engineering educators and by industry, commerce, and government professionals.

[**Professionalism**] *SE 2014 must include exposure to aspects of professional practice as an integral component of the undergraduate curriculum*.

The professional practice of software engineering encompasses a range of issues and activities, including problem solving, project management, ethical and legal concerns, written and oral communication, teamwork, and remaining current in a rapidly changing discipline.

[**Guidance on Implementation**] *SE 2014 must include discussions of strategies and tactics for implementation, along with high-level recommendations.*

Although it is important for SE 2014 to articulate a broad vision of software engineering education, the success of any curriculum depends heavily on implementation details. SE 2014 must provide institutions with advice on the practical concerns of setting up a degree program.

Chapter 4: Overview of Software Engineering Education Knowledge

This chapter describes the body of knowledge that is appropriate for an undergraduate program in software engineering. The knowledge is designated as the Software Engineering Education Knowledge (SEEK).

4.1 Process of Determining the SEEK

The original SEEK described in SE2004 was based on the model used to construct the body of knowledge for computer science in the CCCS volume. Some minor updates have been made to that model, but its basic structure remains the same. A survey was conducted to determine needed improvements, and several workshops and informal discussion sessions were held to collect input from the software engineering community.

4.2 Knowledge Areas, Units, and Topics

Knowledge is a term used to describe the whole spectrum of content for the discipline: information, terminology, artifacts, data, roles, methods, models, procedures, techniques, practices, processes, and literature. The SEEK is organized hierarchically into three levels. The highest level of the hierarchy is the education *knowledge area*, representing a particular subdiscipline of software engineering that is generally recognized as a significant part of the software engineering knowledge that an undergraduate should know. Knowledge areas are high-level structural elements used for organizing, classifying, and describing software engineering knowledge. Each area is identified by an abbreviation, such as PRF for professional practice.

Each area is broken down into smaller divisions called *units*, which represent individual thematic modules within an area. Adding a two- or three-letter suffix to the area identifies each unit—for example, PRF.com is a professional practice unit on communication skills.

Each unit is further subdivided into a set of *topics*, which are the lowest level of the hierarchy.

4.3 Core Material

In determining the SEEK, it is recognized that software engineering, as a discipline, is relatively immature and that common agreement on the definition of an education body of knowledge is evolving. The SEEK developed and presented in this document is based on previous studies and commentaries on the recommended content for the discipline. It was specifically designed to support the development of undergraduate software engineering curricula and, therefore, does not include all the knowledge that would exist in a more generalized body of knowledge representation. Hence, a body of *core* knowledge has been defined. The SEEK core consists of the essential material that

professionals teaching software engineering agree is necessary for anyone to obtain an undergraduate degree in this field. By insisting on a broad consensus on the core's definition, it is hoped the core will be as small as possible, giving institutions the freedom to tailor the elective components of the curriculum in ways that meet individual program needs.

The following points should be emphasized to clarify the relationship between the SEEK and the ultimate goal of providing undergraduate software engineering curriculum recommendations:

- *The core is not a complete curriculum.* Because the core is defined as minimal, it does not, by itself, constitute a complete undergraduate curriculum. Every undergraduate program will include additional units, both within and outside the software engineering body of knowledge, which this document does not attempt to address.
- *Core units should span a student's entire education program.* Although many of the units defined as core are introductory, there are also some core units that clearly must be covered only after students have developed significant background in the field. For example, topics such as project planning and tracking, requirements elicitation, and abstract high-level modeling may require knowledge and sophistication that lower-division students do not possess. Similarly, introductory courses may include elective units (additional material that falls outside the core). The designation "core" simply means *required* and says nothing about the level of the course in which it appears.

4.4 Unit of Time

To ensure consistency with other curriculum reports, the SEEK uses lecture hours, abbreviated to *hours*, to quantify instructional time; this measure is generally understandable in (and transferable to) cross-cultural contexts. Thus, an *hour* corresponds to the time required to present the material in a traditional lecture-oriented format; it does not include any additional work that is associated with a lecture (such as self-study, laboratory sessions, and assessments).

This choice of unit does not require or endorse the use of traditional lectures. Still, the time specifications should serve as a comparative measure in the sense that a five-hour unit will presumably take roughly five times as much time to cover as a one-hour unit, independent of the teaching style.

Students are expected to spend a significant amount of additional time outside of class (approximately two to three times the in-class hours) developing facility with the material presented in class because mastery of some topics requires considerable practice and reflection on their part.

For reference, a 15-week "semester" course with three 50-minute ("one hour") lectures per week would represent approximately 45 hours. The 467 hours of content identified in the SEEK would thus represent about 10 such courses.

4.5 Relationship of the SEEK to the Curriculum

The SEEK does not represent the curriculum, but rather it provides the foundation for the design, implementation, and delivery of the educational units that make up a software engineering curriculum. In particular, the organization and content of the knowledge areas and knowledge units should not be deemed to imply how the knowledge should be organized into education units or activities. The ordering of the knowledge areas, knowledge units, and topics in the SEEK is for the most part arbitrary and is not meant to imply a corresponding arrangement of topics in a curriculum. In the same way, the software engineering practices (such as requirements analysis, architecture, design, construction, and verification) described in the SEEK may be mapped into a variety of different software development processes (such as plan-driven, iterative, and agile). Furthermore, in a four-year undergraduate program structure that is common in the United States, the SEEK's "content hour" total would be equivalent to about one-fourth of the overall curriculum content, leaving adequate time for additional software engineering material that can be tailored to the objectives and student outcomes of a specific program.

4.6 Selection of Knowledge Areas

The SWEBOK Guide provided the starting point for determining knowledge areas of the original SE 2004 SEEK, with adjustments as needed to stress the fundamental principles, knowledge, and practices that underlie the software engineering discipline in a form suitable for undergraduate education. The current SEEK maintains the same essential structure, with modifications to address the continuing evolution of the discipline and to reflect the experience of existing undergraduate software engineering programs.

4.7 SE Education Knowledge Areas

This section describes the 10 knowledge areas that make up the SEEK: computing essentials (CMP), mathematical and engineering fundamentals (FND), professional practice (PRF), software modeling and analysis (MAA), requirements analysis and specification (REQ), software design (DES), software verification & validation (VAV), software process (PRO), software quality (QUA), and security (SEC). The knowledge areas do not include material about continuous mathematics or the natural sciences; the needs in these areas will be discussed in other parts of this volume. For each knowledge area, there is a short description and then a table that delineates the units and topics for that area. Each knowledge unit includes recommended contact hours. For each topic, a Bloom taxonomy level (indicating what capability a graduate should possess) and the topic's relevance (indicating whether the topic is essential or desirable) are designated. Table 1 summarizes the SEEK knowledge areas, with their sets of knowledge units, and lists the minimum number of hours recommended for each area and unit. The relatively small number of hours assigned to the software quality (QUA) and security (SEC) knowledge units reflects that these areas represent crosscutting concerns that are closely linked to topics in other knowledge units. They have been identified separately to increase their visibility and to recognize their importance across the entire extent of the software engineering discipline.

The cognitive skill level for each topic is specified as follows:

- Knowledge (k): Remembering previously learned material. Test observation and recall of information; that is, "bring to mind the appropriate information" (such as dates, events, places, knowledge of major ideas, and mastery of subject matter).
- Comprehension (c): Understanding information and the meaning of material presented. For example, being able to translate knowledge to a new context, interpret facts, compare, contrast, order, group, infer causes, predict consequences, and so forth.
- Application (a): Using learned material in new and concrete situations. For example, using information, methods, concepts, and theories to solve problems requiring the skills or knowledge presented.

A topic's relevance to the core is designated in a similar manner:

- Essential (E): The topic is part of the core.
- Desirable (D): The topic is not part of the core, but it should be included in the core of a particular program if possible; otherwise, it should be considered part of elective materials.

Related topics may differ in their cognitive level and relevance to the core.

KA/KU	Title	Hours		KA/KU	Title	Hours
CMP	**Computing essentials**	**152**		**DES**	**Software design**	**48**
CMP.cf	Computer science foundations	120		DES.con	Design concepts	3
CMP.ct	Construction technologies	20		DES.str	Design strategies	6
CMP.tl	Construction tools	12		DES.ar	Architectural design	12
				DES.hci	Human-computer interaction design	10
				DES.dd	Detailed design	14
				DES.ev	Design evaluation	3
FND	**Mathematical and engineering fundamentals**	**80**		**VAV**	**Software verification and validation**	**37**
FND.mf	Mathematical foundations	50		VAV.fnd	V&V terminology and foundations	5
FND.ef	Engineering foundations for software	22		VAV.rev	Reviews and static analysis	9
FND.ec	Engineering economics for software	8		VAV.tst	Testing	18
				VAV.par	Problem analysis and reporting	5
PRF	**Professional practice**	**29**		**PRO**	**Software process**	**33**
PRF.psy	Group dynamics and psychology	8		PRO.con	Process concepts	3
PRF.com	Communications skills (specific to SE)	15		PRO.imp	Process implementation	8
PRF.pr	Professionalism	6		PRO.pp	Project planning and tracking	8
				PRO.cm	Software configuration management	6
				PRO.evo	Evolution processes and activities	8
MAA	**Software modeling and analysis**	**28**		**QUA**	**Software quality**	**10**
MAA.md	Modeling foundations	8		QUA.cc	Software quality concepts and culture	2
MAA.tm	Types of models	12		QUA.pca	Process assurance	4
MAA.af	Analysis fundamentals	8		QUA.pda	Product assurance	4
REQ	**Requirements analysis and specification**	**30**		**SEC**	**Security**	**20**
REQ.rfd	Requirements fundamentals	6		SEC.sfd	Security fundamentals	4
REQ.er	Eliciting requirements	10		SEC.net	Computer and network security	8
REQ.rsd	Requirements specification and documentation	10		SEC.dev	Developing secure software	8
REQ.rv	Requirements validation	4				

4.8 Computing Essentials

Computing essentials includes the computer science foundations that support software product design and construction. This area also includes knowledge about the transformation of a design into an implementation as well as the techniques and tools used during this process.

Units and Topics

Reference		k,c,a	E,D	Hours
CMP	**Computing essentials**			152
CMP.cf	Computer science foundations			120
CMP.cf.1	Programming fundamentals (control and data, typing, recursion)	a	E	
CMP.cf.2	Algorithms, data structures, and complexity	a	E	
CMP.cf.3	Problem solving techniques	a	E	

CMP.cf.4	Abstraction, use and support for (encapsulation, hierarchy, etc.)	a	E	
CMP.cf.5	Computer organization	c	E	
CMP.cf.6	Basic user human factors (I/O, error messages, and robustness)	c	E	
CMP.cf.7	Basic developer human factors (comments, structure, and readability)	c	E	
CMP.cf.8	Programming language basics	a	E	
CMP.cf.9	Operating system basics	c	E	
CMP.cf.10	Database fundamentals	c	E	
CMP.cf.11	Network protocols	c	E	
CMP.ct	Construction technologies			20
CMP.ct.1	API design and use	a	E	
CMP.ct.2	Code reuse and libraries	a	E	
CMP.ct.3	Object-oriented runtime issues (e.g., polymorphism and dynamic binding)	a	E	
CMP.ct.4	Parameterization and generics	a	E	
CMP.ct.5	Assertions, design by contract, and defensive programming	a	E	
CMP.ct.6	Error handling, exception handling, and fault tolerance	a	E	
CMP.ct.7	State-based and table-driven construction techniques	c	E	
CMP.ct.8	Runtime configuration and internationalization	a	E	
CMP.ct.9	Grammar-based input processing (parsing)	a	E	
CMP.ct.10	Concurrency primitives (e.g., semaphores and monitors)	a	E	
CMP.ct.11	Construction methods for distributed software (e.g., cloud and mobile computing)	a	E	
CMP.ct.12	Constructing hardware/software systems	c	E	
CMP.ct.13	Performance analysis and tuning	k	E	
CMP.tl	Construction tools			12
CMP.tl.1	Development environments	a	E	
CMP.tl.2	User interface frameworks and tools	c	E	
CMP.tl.3	Unit testing tools	c	E	
CMP.tl.4	Profiling and performance analysis tools		D	

4.9 Mathematical and Engineering Fundamentals

The mathematical and engineering fundamentals of software engineering provide theoretical and scientific underpinnings for the construction of software products with desired attributes. These fundamentals support precisely describing software engineering products. They provide the mathematical foundations to model and facilitate reasoning about these products and their interrelations as well as form the basis for a predictable design process. A central theme is engineering design: a decision-making process of iterative nature, in which computing, mathematics, and engineering sciences are applied to deploy available resources efficiently to meet a stated objective.

Units and Topics

Reference		k,c,a	E,D	Hours
FND	**Mathematical and engineering fundamentals**			80
FND.mf	Mathematical foundations			50
FND.mf.1	Functions, relations, and sets	a	E	
FND.mf.2	Basic logic (propositional and predicate)	a	E	
FND.mf.3	Proof techniques (direct, contradiction, and inductive)	a	E	
FND.mf.4	Basics of counting	a	E	
FND.mf.5	Graphs and trees	a	E	

FND.mf.6	Discrete probability	a	E	
FND.mf.7	Finite state machines and regular expressions	c	E	
FND.mf.8	Grammars	c	E	
FND.mf.9	Numerical precision, accuracy, and errors	c	E	
FND.mf.10	Number theory		D	
FND.ef	Engineering foundations for software			22
FND.ef.1	Empirical methods and experimental techniques (e.g., CPU and memory usage measurement)	c	E	
FND.ef.2	Statistical analysis (e.g., simple hypothesis testing, estimating, regression, and correlation.)	a	E	
FND.ef.3	Measurement and metrics	k	E	
FND.ef.4	Systems development (e.g., security, safety, performance, effects of scaling, and feature interaction)	k	E	
FND.ef.5	Engineering design (e.g., formulation of problem, alternative solutions, and feasibility)	c	E	
FND.ef.6	Theory of measurement (e.g., criteria for valid measurement)	c	E	
FND.ec	Engineering economics for software			8
FND.ec.1	Value considerations throughout the software life cycle	k	E	
FND.ec.2	Evaluating cost-effective solutions (e.g., benefits realization, tradeoff analysis, cost analysis, and return on investment)	c	E	

4.10 Professional Practice

Professional practice is concerned with the knowledge, skills, and attitudes that software engineers must possess to practice software engineering professionally, responsibly, and ethically. The study of professional practices includes the areas of technical communication, group dynamics and psychology, and social and professional responsibilities.

Units and Topics

Reference		k,c,a	E,D	Hours
PRF	**Professional practice**			29
PRF.psy	Group dynamics and psychology			8
PRF.psy.1	Dynamics of working in teams and groups	a	E	
PRF.psy.2	Individual cognition (e.g., limits)	k	E	
PRF.psy.3	Cognitive problem complexity	k	E	
PRF.psy.4	Interacting with stakeholders	c	E	
PRF.psy.5	Dealing with uncertainty and ambiguity	k	E	
PRF.psy.6	Dealing with multicultural environments	k	E	
PRF.com	Communications skills (specific to SE)			15
PRF.com.1	Reading, understanding, and summarizing reading (e.g., source code, and documentation)	a	E	
PRF.com.2	Writing (assignments, reports, evaluations, justifications, etc.)	a	E	
PRF.com.3	Team and group communication (both oral and written, email, etc.)	a	E	
PRF.com.4	Presentation skills	a	E	
PRF.pr	Professionalism			6
PRF.pr.1	Accreditation, certification, and licensing	k	E	
PRF.pr.2	Codes of ethics and professional conduct	c	E	
PRF.pr.3	Social, legal, historical, and professional issues and concerns	c	E	
PRF.pr.4	The nature and role of professional societies	k	E	
PRF.pr.5	The nature and role of software engineering standards	k	E	
PRF.pr.6	The economic impact of software	c	E	
PRF.pr.7	Employment contracts	k	E	

4.11 Software Modeling and Analysis

Modeling and analysis can be considered core concepts in any engineering discipline because they are essential to documenting and evaluating design decisions and alternatives.

Units and Topics

Reference		k,c,a	E,D	Hours
MAA	**Software modeling and analysis**			28
MAA.md	Modeling foundations			8
MAA.md.1	Modeling principles (e.g., decomposition, abstraction, generalization, projection/views, and use of formal approaches)	c	E	
MAA.md.2	Preconditions, postconditions, invariants, and design by contract	c	E	
MAA.md.3	Introduction to mathematical models and formal notation	k	E	
MAA.tm	Types of models			12
MAA.tm.1	Information modeling (e.g., entity-relationship modeling and class diagrams)	a	E	
MAA.tm.2	Behavioral modeling (e.g., state diagrams, use case analysis, interaction diagrams, failure modes and effects analysis, and fault tree analysis)	a	E	
MAA.tm.3	Architectural modeling (e.g., architectural patterns and component diagrams)	c	E	
MAA.tm.4	Domain modeling (e.g., domain engineering approaches)	k	E	
MAA.tm.5	Enterprise modeling (e.g., business processes, organizations, goals, and workflow)		D	
MAA.tm.6	Modeling embedded systems (e.g., real-time schedule analysis, and interface protocols)		D	
MAA.af	Analysis fundamentals			8
MAA.af.1	Analyzing form (e.g., completeness, consistency, and robustness)	c	E	
MAA.af.2	Analyzing correctness (e.g., static analysis, simulation, and model checking)	a	E	
MAA.af.3	Analyzing dependability (e.g., failure mode analysis and fault trees)	k	E	
MAA.af.4	Formal analysis (e.g., theorem proving)	k	E	

4.12 Requirements Analysis and Specification

Requirements represent the real-world needs of users, customers, and other stakeholders affected by a system. The construction of requirements includes elicitation and analysis of stakeholders' needs and the creation of an appropriate description of desired system behavior and qualities, along with relevant constraints and assumptions. The grouping of these requirements practices in a single knowledge area is not intended to imply a particular structure or sequence of activities in a software development process.

Units and Topics

Reference		k,c,a	E,D	Hours
REQ	**Requirements analysis and specification**			30
REQ.rfd	Requirements fundamentals			6
REQ.rfd.1	Definition of requirements (e.g., product, project, constraints,	c	E	

	system boundary, external, and internal)			
REQ.rfd.2	Requirements process	c	E	
REQ.rfd.3	Layers/levels of requirements (e.g., needs, goals, user requirements, system requirements, and software requirements)	c	E	
REQ.rfd.4	Requirements characteristics (e.g., testable, unambiguous, consistent, correct, traceable, and priority)	c	E	
REQ.rfd.5	Analyzing quality (nonfunctional) requirements (e.g., safety, security, usability, and performance)	a	E	
REQ.rfd.6	Software requirements in the context of systems engineering	k	E	
REQ.rfd.7	Requirements evolution	c	E	
REQ.rfd.8	Traceability	c	E	
REQ.rfd.9	Prioritization, trade-off analysis, risk analysis, and impact analysis	c	E	
REQ.rfd.10	Requirements management (e.g., consistency management, release planning, and reuse)	k	E	
REQ.rfd.11	Interaction between requirements and architecture	k	E	
REQ.er	Eliciting requirements			10
REQ.er.1	Elicitation sources (e.g., stakeholders, domain experts, and operational and organization environments)	c	E	
REQ.er.2	Elicitation techniques (e.g., interviews, questionnaires/surveys, prototypes, use cases, observation, and participatory techniques)	a	E	
REQ.rsd	Requirements specification and documentation			10
REQ.rsd.1	Requirements documentation basics (e.g., types, audience, structure, quality, attributes, and standards)	k	E	
REQ.rsd.2	Software requirements specification techniques (e.g., plan-driven requirements documentation, decision tables, user stories, and behavioral specifications)	a	E	
REQ.rv	Requirements validation			4
REQ.rv.1	Reviews and inspections	a	E	
REQ.rv.2	Prototyping to validate requirements	k	E	
REQ.rv.3	Acceptance test design	c	E	
REQ.rv.4	Validating product quality attributes	c	E	
REQ.rv.5	Requirements interaction analysis (e.g., feature interaction)	k	E	
REQ.rv.6	Formal requirements analysis		D	

4.13 Software Design

Software design is concerned with issues, techniques, strategies, representations, and patterns used to determine how to implement a component or a system.

Units and Topics

Reference		k,c,a	E,D	Hours
DES	**Software design**			48
DES.con	Design concepts			3
DES.con.1	Definition of design	c	E	
DES.con.2	Fundamental design issues (e.g., persistent data, storage management, and exceptions)	c	E	
DES.con.3	Context of design within multiple software development life cycles	k	E	
DES.con.4	Design principles (information hiding, cohesion, and coupling)	a	E	
DES.con.5	Interactions between design and requirements	c	E	
DES.con.6	Design for quality attributes (e.g., reliability, usability,	k	E	

	maintainability, performance, testability, security, and fault tolerance)			
DES.con.7	Design trade-offs	k	E	
DES.str	Design strategies			6
DES.str.1	Function-oriented design	c	E	
DES.str.2	Object-oriented design	a	E	
DES.str.3	Data-structure centered design		D	
DES.str.4	Aspect-oriented design		D	
DES.ar	Architectural design			12
DES.ar.1	Architectural styles, patterns, and frameworks	a	E	
DES.ar.2	Architectural trade-offs among various attributes	a	E	
DES.ar.3	Hardware and systems engineering issues in software architecture	k	E	
DES.ar.4	Requirements traceability in architecture	k	E	
DES.ar.5	Service-oriented architectures	k	E	
DES.ar.6	Architectures for network, mobile, and embedded systems	k	E	
DES.ar.7	Relationship between product architecture and the structure of development organization and market	k	E	
DES.hci	Human-computer interaction design			10
DES.hci.1	General HCI design principles	a	E	
DES.hci.2	Use of modes and navigation	a	E	
DES.hci.3	Coding techniques and visual design (e.g., color, icons, and fonts)	c	E	
DES.hci.4	Response time and feedback	a	E	
DES.hci.5	Design modalities (e.g., direct manipulation, menu selection, forms, question-answer, and commands)	a	E	
DES.hci.6	Localization and internationalization	c	E	
DES.hci.7	HCI design methods	c	E	
DES.hci.8	Interface modalities (e.g., speech and natural language, audio/video, and tactile)		D	
DES.hci.9	Metaphors and conceptual models		D	
DES.hci.10	Psychology of HCI		D	
DES.dd	Detailed design			14
DES.dd.1	Design patterns	a	E	
DES.dd.2	Database design	a	E	
DES.dd.3	Design of networked and mobile systems	a	E	
DES.dd.4	Design notations (e.g., class and object diagrams, UML, state diagrams, and formal specification)	c	E	
DES.ev	Design evaluation			3
DES.ev.1	Design attributes (e.g., coupling, cohesion, information hiding, and separation of concerns)	k	E	
DES.ev.2	Design metrics	a	E	
DES.ev.3	Formal design analysis		D	

4.14 Software Verification and Validation

Software verification and validation uses a variety of techniques to ensure that a software component or system satisfies its requirements and meets stakeholder expectations.

Units and Topics

Reference		k,c,a	E,D	Hours
VAV	**Software verification and validation**			37
VAV.fnd	V&V terminology and foundations			5
VAV.fnd.1	V&V objectives and constraints	k	E	

VAV.fnd.2	Planning the V&V effort	k	E	
VAV.fnd.3	Documenting V&V strategy, including tests and other artifacts	a	E	
VAV.fnd.4	Metrics and measurement (e.g., reliability, usability, and performance)	k	E	
VAV.fnd.5	V&V involvement at different points in the life cycle	k	E	
VAV.rev	Reviews and static analysis			9
VAV.rev.1	Personal reviews (design, code, etc.)	a	E	
VAV.rev.2	Peer reviews (inspections, walkthroughs, etc.)	a	E	
VAV.rev.3	Static analysis (common defect detection, checking against formal specifications, etc.)	a	E	
VAV.tst	Testing			18
VAV.tst.1	Unit testing and test-driven development	a	E	
VAV.tst.2	Exception handling (testing edge cases and boundary conditions)	a	E	
VAV.tst.3	Coverage analysis and structure-based testing	a	E	
VAV.tst.4	Black-box functional testing techniques	a	E	
VAV.tst.5	Integration testing	c	E	
VAV.tst.6	Developing test cases based on use cases and/or user stories	a	E	
VAV.tst.7	Testing based on operational profiles (e.g., most-used operations first)	k	E	
VAV.tst.8	System and acceptance testing	a	E	
VAV.tst.9	Testing across quality attributes (e.g., usability, security, compatibility, and accessibility)	a	E	
VAV.tst.10	Regression testing	c	E	
VAV.tst.11	Testing tools and automation	a	E	
VAV.tst.12	User interface testing	k	E	
VAV.tst.13	Usability testing	a	E	
VAV.tst.14	Performance testing	k	E	
VAV.par	Problem analysis and reporting			5
VAV.par.1	Analyzing failure reports	c	E	
VAV.par.2	Debugging and fault isolation techniques	a	E	
VAV.par.3	Defect analysis (e.g., identifying product or process root cause for critical defect injection or late detection)	k	E	
VAV.par.4	Problem tracking	c	E	

4.15 Software Process

Software process is concerned with providing appropriate and effective structures for the software engineering practices used to develop and maintain software components and systems at the individual, team, and organizational levels. This knowledge area covers various process models and supports individual and team experiences with one or more software development processes, including planning, execution, tracking, and configuration management.

Units and Topics

Reference		k,c,a	E,D	Hours
PRO	**Software process**			33
PRO.con	Process concepts			3
PRO.con.1	Themes and terminology	k	E	
PRO.con.2	Software engineering process infrastructure (e.g., personnel, tools, and training)	k	E	

PRO.con.3	Modeling and specification of software processes	c	E	
PRO.con.4	Measurement and analysis of software processes	c	E	
PRO.con.5	Software engineering process improvement (individual, team, and organization)	c	E	
PRO.con.6	Quality analysis and control (e.g., defect prevention, review processes, quality metrics, and root cause analysis of critical defects to improve processes and practices)	c	E	
PRO.con.7	Systems engineering life-cycle models		D	
PRO.imp	Process implementation			8
PRO.imp.1	Levels of process definition (e.g., organization, project, team, and individual)	k	E	
PRO.imp.2	Life-cycle model characteristics (e.g., plan-based, incremental, iterative, and agile)	c	E	
PRO.imp.3	Individual software process (model, definition, measurement, analysis, and improvement)	a	E	
PRO.imp.4	Team process (model, definition, organization, measurement, analysis, and improvement)	a	E	
PRO.imp.5	Software process implementation in the context of systems engineering	k	E	
PRO.imp.6	Process tailoring	k	E	
PRO.imp.7	Effect of external factors (e.g., contract and legal requirements, standards, and acquisition practices) on software process	k	E	
PRO.pp	Project planning and tracking			8
PRO.pp.1	Requirements management (e.g., product backlog, priorities, dependencies, and changes)	a	E	
PRO.pp.2	Effort estimation (e.g., use of historical data and consensus-based estimation techniques)	a	E	
PRO.pp.3	Work breakdown and task scheduling	a	E	
PRO.pp.4	Resource allocation	c	E	
PRO.pp.5	Risk management (e.g., identification, mitigation, remediation, and status tracking)	a	E	
PRO.pp.6	Project tracking metrics and techniques (e.g., earned value, velocity, burndown charts, defect tracking, and management of technical debt)	a	E	
PRO.pp.7	Team self-management (e.g., progress tracking, dynamic workload allocation, and response to emergent issues)	a	E	
PRO.cm	Software configuration management			6
PRO.cm.1	Revision control	a	E	
PRO.cm.2	Release management	c	E	
PRO.cm.3	Configuration management tools	c	E	
PRO.cm.4	Build processes and tools, including automated testing and continuous integration	a	E	
PRO.cm.5	Software configuration management processes	k	E	
PRO.cm.6	Maintenance issues	k	E	
PRO.cm.7	Distribution and backup		D	
PRO.evo	Evolution processes and activities			8
PRO.evo.1	Basic concepts of evolution and maintenance	k	E	
PRO.evo.2	Working with legacy systems	k	E	
PRO.evo.3	Refactoring	c	E	

4.16 Software Quality

Software quality is a crosscutting concern, identified as a separate entity to recognize its importance and provide a context for achieving and ensuring quality in all aspects of

software engineering practice and process. These software quality topics must therefore be integrated into the presentation and application of material associated with other knowledge areas.

Units and Topics

Reference		k,c,a	E,D	Hours
QUA	**Software quality**			10
QUA.cc	Software quality concepts and culture			2
QUA.cc.1	Definitions of quality	k	E	
QUA.cc.2	Society's concern for quality	k	E	
QUA.cc.3	The costs and impacts of bad quality	k	E	
QUA.cc.4	A cost of quality model	c	E	
QUA.cc.5	Quality attributes for software (e.g., dependability, usability, and safety)	k	E	
QUA.cc.6	Roles of people, processes, methods, tools, and technology	k	E	
QUA.pca	Process assurance			4
QUA.pca.1	The nature of process assurance	k	E	
QUA.pca.2	Quality planning	k	E	
QUA.pca.3	Process assurance techniques	k	E	
QUA.pda	Product assurance			4
QUA.pda.1	The nature of product assurance	k	E	
QUA.pda.2	Distinctions between assurance and V&V	k	E	
QUA.pda.3	Quality product models	k	E	
QUA.pda.4	Root cause analysis and defect prevention	c	E	
QUA.pda.5	Quality product metrics and measurement	c	E	
QUA.pda.6	Assessment of product quality attributes (e.g., usability, reliability, and availability)	c	E	

4.17 Security

Software security has two distinct but related components. As a standalone knowledge area, it deals with the protection of information, systems, and networks. As a crosscutting concern, it provides a focus on how security must be incorporated into all parts of the software development life cycle. To prepare software engineers who can develop secure software, security must be integrated with the practices and processes associated with other knowledge areas.

Units and Topics

Reference		k,c,a	E,D	Hours
SEC	**Security**			20
SEC.sfd	Security fundamentals			4
SEC.sfd.1	Information assurance concepts (confidentiality, integrity, and availability)	k	E	
SEC.sfd.2	Nature of threats (e.g., natural, intentional, and accidental)	k	E	
SEC.sfd.3	Encryption, digital signatures, message authentication, and hash functions	c	E	
SEC.sfd.4	Common cryptographic protocols (applications, strengths, and weaknesses)	c	E	
SEC.sfd.5	Nontechnical security issues (e.g., social engineering)	c	E	
SEC.net	Computer and network security			8
SEC.net.1	Network security threats and attacks	k	E	
SEC.net.2	Use of cryptography for network security	k	E	

SEC.net.3	Protection and defense mechanisms and tools	c	E	
SEC.dev	Developing secure software			8
SEC.dev.1	Building security into the software development life cycle	c	E	
SEC.dev.2	Security in requirements analysis and specification	a	E	
SEC.dev.3	Secure design principles and patterns	a	E	
SEC.dev.4	Secure software construction techniques	a	E	
SEC.dev.5	Security-related verification and validation	a	E	

Chapter 5: Guidelines for SE Curriculum Design and Delivery

Chapter 4 of this document presents the SEEK, which identifies the knowledge that software engineering graduates need to learn. However, *how* the SEEK topics should be taught may be as important as *what* is taught. In this chapter, a series of guidelines are described that should be considered by those developing an undergraduate SE curriculum and by those teaching individual SE courses.

5.1 Developing and Teaching the Curriculum

Curriculum Guideline 1: Curriculum designers and instructors must have sufficient relevant knowledge and experience, and understand the character of software engineering.

Curriculum designers and instructors should have engaged in scholarship in the broad area of software engineering. This implies

- having software engineering knowledge in most areas of the SEEK;
- obtaining real-world experience in software engineering;
- becoming recognized publicly as knowledgeable in software engineering either by having a track record of publication or being active in an appropriate professional society;
- being exposed to the continually expanding variety of domains of application of software engineering (such as other branches of engineering or business applications), while being careful not to claim to be experts in those domains; and
- possessing the motivation and the means to keep up to date with developments in the discipline.

Failure to adhere to this principle will open up a program or course to certain risks:

- A program or course might be biased excessively to one kind of software or class of methods, thus failing to give students a broad exposure to or an accurate perception of the field. For example, instructors who have experienced only real-time or only data-processing systems are at risk of flavoring their programs excessively toward these types of systems. Although it is not bad to have programs that are specialized toward specific types of software engineering, these specializations should be explicitly acknowledged in course titles. Also, in a program as a whole, students should eventually be exposed to a comprehensive selection of systems and approaches.
- Faculty members who have a primarily theoretical computer science background might not adequately convey to students the engineering-oriented aspects of software engineering.
- Faculty members from related branches of engineering might deliver a software engineering program or course without a full appreciation of the computer science fundamentals that underlie so much of what software engineers do. They also might

not cover the wide range of domains beyond engineering to which software engineering can be applied.

- Faculty members who have not experienced the development of large systems might not appreciate the importance of process, quality, and security (which are knowledge areas of the SEEK).
- Faculty members who have made a research career out of pushing the frontiers of software development might not appreciate that students first need to be taught what they can use in practice and that they need to understand both practical and theoretical motivations behind what they are taught.

5.2 Constructing the Curriculum

Curriculum Guideline 2: Curriculum designers and instructors must think in terms of outcomes.

Both entire programs and individual courses should include attention to outcomes or learning objectives. Furthermore, as courses are taught, these outcomes should be regularly kept in mind. Thinking in terms of outcomes helps ensure that the material included in the curriculum is relevant and taught in an appropriate manner and at an appropriate level of depth.

The student learning outcomes (see Chapter 3) should be used as a basis for designing and assessing software engineering curricula in general. These can be further specified for the design of individual courses.

In addition, particular institutions may develop more specialized outcomes (e.g., particular abilities in selected applications areas or deeper abilities in certain SEEK knowledge areas).

Curriculum Guideline 3: Curriculum designers must strike an appropriate balance between material coverage and the flexibility to allow for innovation.

There is a tendency among those involved in curriculum design to fill up a program or course with extensive lists of things that "absolutely must" be covered, leaving relatively little time for flexibility or deeper (but less broad) coverage.

However, there is also a strong body of opinion that students who are given a foundation in the "basics" and an awareness of advanced material should be able to fill in many "gaps" in their education after graduation on an as-needed basis. This suggests that certain kinds of advanced process-oriented SEEK material, although marked at an "a" (application) level of coverage, could be covered at a "k" level if absolutely necessary to allow for various sorts of curriculum innovation. Nevertheless, material with deeper technical or mathematical content marked "a" should not be reduced to "k" coverage because it tends to be much harder to learn on the job.

Curriculum Guideline 4: Many SE concepts, principles, and issues should be taught as recurring themes throughout the curriculum to help students develop a software engineering mindset.

Material defined in many SEEK units should be taught in a manner that is distributed throughout many courses in the curriculum. Generally, early courses should introduce the material, with subsequent courses reinforcing and expanding upon the material. In most cases, there should also be courses, or parts of courses, that treat the material in depth.

In addition to ethics and tool use, which will be highlighted specifically in other guidelines, the following are types of material that should be presented, at least in part, as recurring themes:

- Measurement, quantification, and formal or mathematical approaches.
- Modeling, representation, and abstraction.
- Human factors and usability: Students need to repeatedly see how software engineering is not just about technology.
- The fact that many software engineering principles are in fact core engineering principles: Students may learn SE principles better if they are shown examples of the same principle in action elsewhere—for example, the fact that all engineers use models, measure, solve problems, and use "black boxes."
- The importance of scale: Students can practice only on relatively small problems, yet they need to appreciate that the power of many techniques is most obvious in large systems. They need to be able to practice tasks as if they were working on very large systems and to practice reading, understanding, and making small changes to large systems.
- The importance of reuse.
- Much of the material in the process and quality knowledge areas.
- Reflection and evaluation: Students should assess their ideas through a range of forms such as concept analysis, concept implementation, and empirical studies.

Curriculum Guideline 5: Learning certain software engineering topics requires maturity, so these topics should be taught toward the end of the curriculum, while other material should be taught earlier to facilitate gaining that maturity.

It is important to structure the material that has to be taught in such a way that students fully appreciate the underlying principles and the motivation. If taught too early in the curriculum, many topics from SEEK's process and quality knowledge areas are likely to be poorly understood and poorly appreciated by students. This should be taken into account when designing the sequence in which material is taught and how real-world experiences are introduced to the students. That is, introductory material on these topics should be taught in early years, but the bulk of the material should be left until the latter part of the curriculum.

On the other hand, students also need practical material to be taught early so they can begin to gain maturity by participating in real-world development experiences (such as internships, cooperative education, open source project participation, or student projects).

For example, topics that should be taught early include programming, human factors, aspects of requirements and design, and verification and validation. This does not mean to imply that programming must be taught first, but a reasonable amount should be taught in a student's first year.

Students should also be exposed to "difficult" software engineering situations relatively early in their program. Examples of these might be dealing with rapidly changing requirements, having to understand and change a large existing system, having to work in a large team, and so forth. The goal of such experiences is to raise awareness in students that process, quality, and security are important things to study, *before* they start studying them.

Curriculum Guideline 6: Students should develop an understanding of a software application domain.

Almost all software engineering activity will involve solving problems for clients in domains beyond software engineering itself. Therefore, somewhere in the curriculum, students should be able to study one or more application domains in reasonable depth. Studying such material will give students direct domain knowledge they can apply to software engineering problems and will also teach them the domain's language and thought processes, enabling more in-depth study later on.

The choice of domain (or domains) is a local consideration and, in many cases, may be left up to the student. Domains can include other branches of engineering, the natural sciences, social sciences, business, and the humanities. No one domain should be considered more important to software engineering programs than another. The study of certain domains may necessitate additional supporting courses, such as particular areas of mathematics and computer science as well as deeper areas of software engineering.

This guideline does not preclude the possibility of designing courses or programs that deeply integrate the teaching of domain knowledge with the teaching of software engineering. In fact, such an approach would be innovative. For example, an institution could have courses called "Telecommunications Software Engineering," "Aerospace Software Engineering," "Information Systems Software Engineering," or "Software Engineering of Sound and Music Systems." However, in such cases, great care must be taken to ensure that depth is not sacrificed in either SE or the domain. The risk is that the instructor, the instructional material, or the presentation may not have adequate depth in one or the other area.

5.3 Attributes and Attitudes That Should Pervade the Curriculum and Its Delivery

Curriculum Guideline 7: Software engineering must be taught in ways that recognize it is both a computing and an engineering discipline.

Educators should develop an appreciation of the aspects of software engineering that it shares with other branches of engineering and with other branches of computing, particularly computer science. Characteristics of engineering and computing are presented in Chapter 2.

- **Engineering**: Engineering has been evolving for millennia, and a great deal of general knowledge has been built up, although much of it needs to be adapted to the software engineering context. Software engineering students must understand their roles as engineers and develop a sense of engineering practice. This can be achieved only by appropriate attitudes on the part of all faculty and administrators.

  Because software engineering differs from other engineering disciplines in the nature of its products, processes, and underlying science, students should be prepared to communicate those differences to other engineers, while at the same time having a solid understanding of how their own work fits into the broader engineering profession.

- **Computing**: For software engineers to have the technical competence to develop high-quality software, they must have a solid and deep background in the fundamentals of computer science, as outlined in Chapter 4. That knowledge will ensure they understand both the limits of computing and the technologies available to undertake a software engineering project.

  This principle does not require that a software engineer's knowledge of these areas be as deep as a computer scientist's. However, the software engineer needs to have sufficient knowledge and practice to choose from among and apply these technologies appropriately. Software engineers also must have sufficient appreciation of the complexity of these technologies to recognize when they are beyond their area of expertise and therefore need to consult a specialist.

Curriculum Guideline 8: Students should be trained in certain personal skills that transcend the subject matter.

The following skills tend to be required for almost all activities that students will encounter in the workforce. These skills must be acquired primarily through practice:

- **Exercising critical judgment**: Assessing competing solutions is a key part of what it means to be an engineer. Curriculum design and delivery should therefore help students build the knowledge, analysis skills, and methods they need to make sound judgments. Of particular importance is a willingness to think critically. Students should also be taught to judge the reliability of various sources of information.
- **Evaluating and challenging received wisdom:** Students should be trained to not immediately accept everything they are taught or read. They should also gain an understanding of the limitations of current SE knowledge and how SE knowledge seems to be developing.
- **Recognizing their own limitations**: Students should be taught that professionals consult other professionals and that there is great strength in teamwork.
- **Communicating effectively**: Students should learn to communicate well in all contexts: in writing, when giving presentations, when demonstrating (their own or

others') software, and when conducting discussions with others. Students should also build listening, cooperation, and negotiation skills.

- **Behaving ethically and professionally:** Students should learn to think about the ethical, privacy, and security implications of their work. See also Curriculum Guideline 15.

There are some SEEK topics relevant to these points that can be taught in lectures, especially aspects of communication ability; however, students will learn these skills most effectively if they are constantly emphasized though group projects, carefully marked written work, and student presentations.

Curriculum Guideline 9: Students should develop an appreciation of the importance of continued learning and skills for self-directed learning.

Because so much of what is learned will change over a student's professional career and only a small fraction of what could be learned will be taught and learned at university, it is of paramount importance that students develop the habit of continually expanding their knowledge.

Curriculum Guideline 10: Software engineering problem solving should be taught as having multiple dimensions.

An important goal of most software projects is meeting client needs, both explicitly and implicitly. It is important to recognize this when designing programs and courses. Such recognition focuses learners on the rationale for what they are learning, deepens the understanding of the knowledge learned, and helps ensure that the material taught is relevant.

Meeting client needs requires that students learn to solve many types of problems. The curriculum should emphasize the overall goal of providing software that is useful and help students move beyond the technical problems that they tend to be drawn to first. Students should learn to think about and solve problems such as analysis, design, and testing that are related directly to solving the clients' problem. They also need to address meta-problems, such as process improvement, the solutions of which will facilitate product-oriented problem solving. Finally, the curriculum should address areas such as ethical problems that are orthogonal to the other categories.

Problem solving is best learned through practice and taught through examples.

Curriculum Guideline 11: The underlying and enduring principles of software engineering should be emphasized, rather than the details of the latest or specific tools.

The SEEK lists many topics that can be taught using a variety of computer hardware, software applications, technologies, and processes (collectively referred to as tools). In a good curriculum, it is the enduring knowledge in the SEEK topics that must be emphasized, not the details of the tools. The topics are supposed to remain valid for many years; as much as possible, the knowledge and experience derived from their learning should still be applicable 10 or 20 years later. Particular tools, on the other hand, will rapidly change. It is a mistake, for example, to focus excessively on how to use a

particular vendor's piece of software, the detailed steps of a methodology, or the syntax of a specific programming language.

Applying this guideline to languages requires understanding that the line between what is enduring and what is temporary can be somewhat hard to pinpoint because it is a moving target. For example, software engineers should definitely learn several programming languages in detail. This guideline should be interpreted as saying that, when learning such languages, students must learn more than just surface syntax and, having learned the languages, should be able to learn with little difficulty whatever new languages appear.

Applying this guideline to processes (also known as methods or methodologies) is similar to applying it to languages. Rather than memorizing the details of a particular process model, students should be helped to understand the goals being sought and the problems being addressed so they can appropriately evaluate, choose, and adapt processes to support their future work as software engineering professionals.

Applying this guideline to technologies (both hardware and software) means that students should develop skills in using documentation and other resources to acquire the knowledge needed to work effectively with previously unfamiliar components and systems, rather than being taught only the details of a particular technology.

Curriculum Guideline 12: The curriculum must be taught so that students gain experience using appropriate and up-to-date tools, even though tool details are not the focus of the learning.

Performing software engineering efficiently and effectively requires choosing and using the most appropriate computer hardware, software tools, technologies, and processes (collectively referred to here as tools). Students must develop skill in choosing and using tools so they go into the workforce with a habit of working with tools and an understanding that selecting and developing facility with tools is a normal part of professional work.

Appropriateness of tools must be carefully considered. Tool selection should consider complexity, reliability, expense, learning curve, functionality, and benefit. Tool selection also needs to consider educational value and usefulness in the workplace after graduation. Open source tools are often an attractive option given the reduced costs for students and the prominent market presence of many open source tools in the workplace.

Tools used in curricula must be reasonably up to date for several reasons: (a) students can take the tools into the workplace as "ambassadors," performing a form of technology transfer; (b) students can take advantage of the tool skills they have learned; (c) and students and their employers will not feel the education is out of date. Having said that, older tools can sometimes be simpler and therefore more appropriate for certain needs.

This guideline may seem to conflict with CG 11, but that conflict is illusory. The key to understanding these guidelines is to recognize that CG 11 identifies the fundamental emphasis on the principles of software engineering. Having established that emphasis, CG 12 recognizes that selecting, learning about, and using tools is an essential part of

professional software engineering. Students must develop the relevant skills and understand the role of tools in the same way that they need to develop skills in programming.

Curriculum Guideline 13: Material taught in a software engineering program should, where possible, be grounded in (a) sound empirical research and mathematical or scientific theory or (b) widely accepted good practice.

There must be evidence that whatever is taught is true and useful. This evidence can take the form of validated scientific or mathematical theory (such as in many areas of computer science), systematically gathered empirical evidence, or widely used and generally accepted best practice.

It is important, however, not to be overly dogmatic about the application of a particular theory because its use may not always be appropriate. For example, formalizing a specification or design in order to apply mathematical approaches can be inefficient and reduce agility in many situations. In other circumstances, however, it may be essential.

In situations where the material taught is based on generally accepted practice that has not yet been scientifically validated, it should be made clear to students that the material is still open to question.

When teaching "good practices," they should not be presented in a context-free manner; examples of the success of the practices and of failure caused by not following them should be included. The same should be true when presenting knowledge derived from research.

This guideline complements CG 11. Whereas CG 11 stresses focus on fundamental software engineering principles, CG 13 says that what is taught should be well founded.

Curriculum Guideline 14: The curriculum should have a significant real-world basis.

Incorporating real-world elements into the curriculum is necessary to enable effective learning of software engineering skills and concepts. A program should incorporate at least some of the following:

- **Case studies**: Exposure to real systems and project case studies is important, with students taught to critique these examples and to reuse the best parts.
- **Project-based activities**: Some learning activities should be set up to mimic typical projects in industry. These should include group work, presentations, formal reviews, quality assurance, and so forth. It can be beneficial to include real-world stakeholders or interdisciplinary teams. Students should understand and be able to experience the various roles typically found in a contemporary software engineering team.
- **Capstone project**: Students should complete a significant project, preferably spanning their entire last year, in order to practice the knowledge and skills they have learned. Building on skills developed in other project-based learning activities, students should be given the primary responsibility to manage this capstone project, which is further discussed in Section 6.2. Team projects are most common and

considered to be best practice because students can develop team skills that have value in many professional environments.

- **Practical exercises**: Students should be given practical exercises so they can develop skills in current practices and processes.
- **Student work experience**: Where possible, students should have some form of industrial work experience as a part of their program. The terminology for this is country-dependent but includes internships, cooperative education, and sandwich work terms. The intent is to provide significant experience with software products developed by teams that have real stakeholders including active users, a large code base, ongoing development, packaging and distribution cycles, and product documentation. If opportunities for work experience are difficult to provide, courses must simulate these experiences to the extent possible. Student participation in open source projects is another possible approach to providing these experiences.

Despite these guidelines, instructors should keep in mind that the level of real-world exposure their students can achieve as an undergraduate will be limited; students will generally come to appreciate the extreme complexity and true consequences of poor work only through experience as they work on various projects in their careers. Educators can only start the process of helping students develop a mature understanding of the real world, and educators must realize that it will be difficult to enable students to appreciate everything they are taught.

Curriculum Guideline 15: Ethical, legal, and economic concerns and the notion of what it means to be a professional should be raised frequently.

One of the key reasons for the existence of a defined profession is to ensure that its members follow ethical and professional principles. If students discuss these issues throughout the curriculum, they will become deeply entrenched. One aspect of this is exposing students to standards and guidelines. See Section 2.4 for further discussion of professionalism.

Curriculum Guideline 16: Software process should be central to the curriculum organization and to students' understanding of software engineering practice.

Software process is both a focal and crosscutting topic in every software engineering degree program. Software process is also one of the most popular and sometimes contentious topics encountered in the discussion and feedback related to this document. There are many comments elsewhere in this chapter that are relevant to software process, its role in the curriculum, and how to teach it, but this guideline attempts to pull much of that together and add comments particular to process.

Evolution of software process best practice: It is important to note that this curriculum guideline does not endorse any particular software process. Software process has evolved over the years, and it is reasonable to expect that this will continue. Assuming that one particular process or style of process is the best or final answer seems akin to making a similar assumption about a particular programming language or operating system. Every curriculum, while covering software process in depth, should also give students an

appreciation of the range of processes and the notion that best practice in this area has and will continue to change.

Range of software processes: Addressing the range of software processes implies that the curriculum give students some understanding of a selection of processes that might include, for example, both plan-based and agile methods. Some of this material could be covered as a survey of processes. The selection of processes to cover should reflect current industry practice.

Motivating software process: Like many aspects of software engineering, process is difficult to motivate until students understand central challenges such as scale, complexity, and human communication that motivate all of software engineering. This has two implications for designing the curriculum. First, process needs to be introduced gradually. Making software process part of student work early in the curriculum helps to develop good habits. But process use must be carried through to later courses when student appreciation for process has been developed. Second, student work must include exposure to larger systems to develop an appreciation of the challenges that motivate software engineering. Early and repeated exposure to larger systems is desirable. This might include not only student team project work but also case studies and observation of working systems. Industry experience such as internships may provide these experiences, and the availability of open source projects also provides a source of materials.

Software process in context: The curriculum should address the relationship of software process and other elements of a work environment. For example, the supportive (or limiting) role of software development tools in successful processes use should be addressed as part of the coverage of tools. Students also need to learn about the importance of organizational culture, team and product size, and application domain in process selection. Environmental considerations such as these provide the context needed for students to understand the range of processes. Examples might include the continued presence of plan-based processes in the development of large embedded hardware/software systems or the advantages of agile processes in domains where requirements are incompletely understood or rapidly changing.

Depth and application of software process: An appreciation of the range of processes should be combined with the development of the skills and knowledge to apply at least one particular process. Programs may need to focus on one particular process for students to achieve any proficiency by graduation. Even basic proficiency will only develop through repeated exposure across the curriculum, including student use of process in their own project work, team-based projects, and projects of sufficient scale to make process use meaningful.

Process improvement: Addressing the range and context of software process provides a foundation for students to learn that software processes are not static, but rather they are something to be selected, managed, and improved. The curriculum should build on this foundation and directly address concepts of process improvement. Doing so requires students to understand process as an entity and an example of abstraction. Making that

leap opens software process to concepts of modeling, analysis, measurement, and design that are central to process improvement.

5.4 General Strategies for Software Engineering Pedagogy

Curriculum Guideline 17: To ensure that students embrace certain important ideas, care must be taken to motivate students by using interesting, concrete, and convincing examples.

Some concepts and techniques considered central to the software engineering discipline are only learned through bitter experience. In some cases, this is because the educational community has not appreciated and taught the value of such concepts. In other cases, educators have encountered skepticism on the part of students.

Thus, there is a need to put considerable attention into motivating students to accept ideas by using interesting, concrete, and revealing examples. The examples should be of sufficient size and complexity so as to demonstrate that using the material being taught has obvious benefits and that failure to use the material could lead to undesirable consequences.

The following are examples of areas where motivation is particularly needed:

- *Mathematical foundations*: Logic and discrete mathematics should be taught in the context of their application to software engineering or computer science problems. If derivations and proofs are presented, these should preferably be taught following an explanation of why the result is important. Statistics and empirical methods should likewise be taught in an applied, rather than an abstract, manner.
- *Process and quality*: Students must be made aware of the consequences of poor processes and bad quality. They must also be exposed to good processes and quality so they can experience for themselves the effect of improvements, feel pride in their work, and learn to appreciate good work.
- *Human factors and usability*: Students will often not appreciate the need for attention to these areas unless they actually experience usability difficulties or watch users having difficulty using software.

Curriculum Guideline 18: Software engineering education needs to move beyond the lecture format and to consider a variety of teaching and learning approaches.

The most common approach to teaching software engineering material is the use of lectures, supplemented by laboratory sessions, tutorials, and so on. However, alternative approaches can help students learn more effectively. Central to designing learning activities for software engineering is recognition of the need for students to participate in time-limited, iterative development experiences. In addition to reflecting common industry practice, iterations are important to motivating student learning. Iterating on prior work helps students see the deficiencies of their efforts in prior iterations and provides an opportunity for reflection and improvement that would otherwise be unavailable.

The following general pedagogical approaches might be considered to supplement or even largely replace the lecture format in certain cases:

- *Problem-based learning*: This has been found to be particularly useful in other professional disciplines and is now used to teach engineering in some institutions. See CG 10 for a discussion of the discipline's problem-solving nature.
- *Just-in-time learning*: Teaching fundamental material immediately before teaching the application of that material. For example, an instructor might teach aspects of mathematics the day before they are applied in a software engineering context. There is evidence that this helps students retain fundamental material, although the approach can be difficult to implement because faculty members must coordinate content across courses.
- *Learning by failure*: Students are given a task that they will have difficulty with. They are then taught methods that would enable them to do the task more easily in the future.
- *Technology-enhanced learning*: The options for individual and team-learning activities enabled by technology continue to expand and evolve. These include simulations, open education resources, intelligent tutoring, quiz and practice systems, and products that support distributed coordination and collaboration.

Curriculum Guideline 19: Important efficiencies and synergies can be achieved by designing curricula so that several types of knowledge are learned at the same time.

As some reviewers have noted, the SEEK identifies a numerous topics, a number of which have been assigned a rather small number of hours. With careful attention to curricular design, however, many topics can be taught concurrently. It is often the case that two topics listed that require x and y hours respectively may be taught together in less than $(x + y)$ hours.

For example, this kind of synergistic teaching and learning may be applied in the following cases:

- *Modeling, languages, and notations*: Familiarity with a modeling notation such as UML can be achieved by using it to illustrate and explain other concepts. The same applies to formal methods and programming. Clearly, there will need to be some time set aside to teach the basics of a language or modeling technique per se, but both broad and deep knowledge can be learned as students study a range of other topics.
- *Process, quality, and tools*: Students can be instructed to follow certain processes, use tools, and include quality assurance activities as they are working on exercises or projects when the explicit objective is to learn other concepts. In these circumstances, it would be desirable for students to have had some prior introduction to relevant processes, tools, or QA techniques so that they know why they are being asked to include them. The learning could be reinforced by following the exercise or project with a discussion of the usefulness of applying the particular technique or tool. The depth of learning of the process is likely to be considerable, with relatively little time being taken away from the other material being taught.

- *Mathematics*: Students might deepen and expand their understanding of statistics while analyzing some data resulting from reliability or performance studies. Opportunities to deepen understanding of logic and other branches of discrete mathematics also abound.

Teaching multiple concepts at the same time in this manner can, in fact, help students appreciate links among topics and can make material more interesting to them. In both cases, this should lead to better retention of material.

Curriculum Guideline 20: Courses and curricula must be reviewed and updated regularly.

Software engineering is rapidly evolving; hence, most (if not all) courses or curricula will, over time, become out of date. Institutions and instructors must therefore regularly review their courses and programs and make whatever changes are necessary. Although this guideline applies to individual software engineering curricula or courses, the principles enunciated in Section 3.2 make clear that the curriculum guidelines described in this volume must also be the subject of ongoing evolution and renewal.

5.5 Concluding Comment

Although intended to apply to the development of any high-quality software engineering program, these guidelines may not address all relevant concerns. For each institution, there are likely to be local and national needs driven by industry, government, and other constituencies. The aspirations of the students themselves also must be considered. Students must see value in their educational experience as it relates to their career goals, which they are likely to judge by what they are able to learn and to achieve during their time in the program. Certainly, they should feel confident about being able to compete internationally, within the global workforce.

Any software engineering curriculum or syllabus needs to integrate all these various considerations into a single, coherent program. Ideally, a uniform and consistent ethos should permeate individual classes and the environment in which the program is delivered. A software engineering program should instill in the student a set of expectations and values associated with engineering high-quality software systems.

Chapter 6: Designing an Undergraduate Degree Program

The preceding two chapters have identified the topics a software engineering undergraduate curriculum should cover (Chapter 4) and provided guidelines about how this might be organized and presented (Chapter 5). This chapter draws upon these to consider how the material might be organized within a degree program.

Developing a degree program is of course a classic example of a design task. There are many constraints and a need to make trade-offs among different factors; there are no right or wrong solutions, only ones that are better or worse with respect to specific criteria. What is more, much of this is specific to the context of a particular institution, its ethos, and the type of students that it recruits. There are some obvious parallels with software design in the way that a curriculum has both static aspects (the mapping of topics to modules) and dynamic ones (how these are to be delivered).

This chapter begins by identifying some of the key factors and how they might influence design choices. There is also a brief review of some ideas about how a curriculum might be organized. At the end of each issue reviewed in this chapter, the relevant curriculum guidelines are referenced using a table. (Although many guidelines address more than one of these themes, each guideline is associated here with the theme judged most relevant.)

6.1 Factors to Consider When Designing a Degree Program

The extent to which each of these factors will influence decisions will need to be decided locally. However, all need to be considered in some way.

The Stakeholders

The major stakeholders in the degree program will include at least the following:

- **Students**. A curriculum needs to reflect the students' needs in many ways. They are likely to expect a program that will prepare them for employment and/or research in terms of both their knowledge and skills, as discussed in Section 3.1. Students also enter a degree program with an educational background that will influence how the curriculum is organized and presented and that is likely to be specific to the context of a particular country. A degree program also needs to provide an intellectual challenge together with a sense of excitement about the discipline.
- **Teaching staff**. Not only do instructors need to motivate, interest, and encourage students, they also have to provide expert knowledge where appropriate. The need for employing people who are able to address these needs was identified in Chapters 2 and 5.

- **Institution**. Each university or college has its own ethos and structures. At one level, this might influence the type of applications that are used to illustrate teaching; at another it will determine the size of a course or module as well as the ways that it might be assessed. It may also be necessary to offer a software engineering degree program as part of a "package" of programs, sharing some courses and resources.
- **Professional bodies**. Professional organizations play a major role in setting standards in the accreditation of degree programs as well as in long-term (post-degree) development through the provision of continuing professional development (CPD) opportunities and programs.

Curriculum plans are normally documented in a way this is determined by the individual institution and that is likely to encourage attention to the needs of most of these stakeholders as well as other issues such as assessment.

CG	Summary of Guideline
CG 2	Curriculum designers and instructors must think in terms of outcomes.
CG 9	Students should develop an appreciation of the importance of continued learning and skills for self-directed learning.

The Curriculum Material

As emphasized in Chapter 4 and elsewhere, the SEEK identifies knowledge areas and knowledge units in a topic-oriented manner that is not meant to form a blueprint for mapping to courses. Indeed, many topics may need to be covered at more than one level or from more than one perspective. For example, the discussion of professional issues may occur when teaching a range of topics. Similarly, a given course may incorporate a set of knowledge units taken from different knowledge areas.

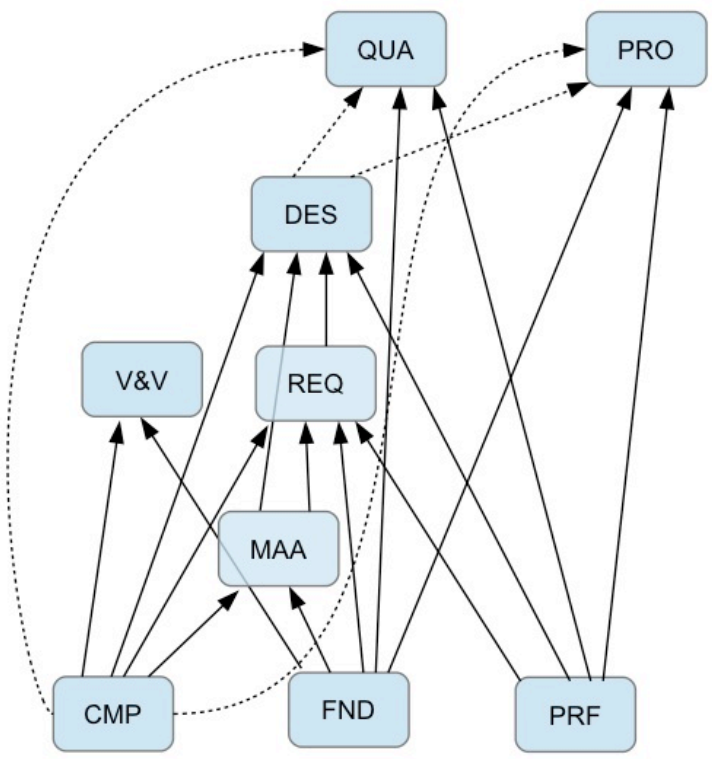

Figure 6.1 Dependencies among Knowledge Areas

A key issue is that of the dependency among topics. A fairly fundamental example is that most software engineering topics implicitly require a basic understanding of programming and of some of the ways that software can be organized (architecture). Others, such as the knowledge units making up software design, and many of the ideas about quality, tend to require a significant level of maturity of understanding together with a breadth of knowledge about many topics if they are to be taught at any depth.

Figure 6.1 shows a basic dependency diagram between the major knowledge areas, with the rationale for the links shown in Table 6.2 The dashed lines linking to QUA and PRO are intended to indicate that the necessary background material might be provided from either DES or CMP, depending on how the material for the latter is delivered. This is not meant to be prescriptive because there may well be knowledge units within an area that are not particularly dependent upon knowledge of another topic. However, it is intended to form a basic indicator of how teaching related to these areas may need to be organized.

	MAA	V&V	REQ	DES	PRO	QUA
CMP	An understanding of the characteristics of software and processes.	An understanding of error types and their causation factors.	Behavior of systems and their interactions.	An understanding of software properties and architecture.	An understanding of the nature of software development activities.	Knowledge about software structures and organization.
FND	Relationships, metrics, structures, etc.	Relationship between tests and predictions.	Descriptions of forms and relationships and possible trade-offs.		Planning needs a measurement basis.	Knowledge needed for formulation of models for product quality.

	MAA	V&V	REQ	DES	PRO	QUA
PRF			Stakeholder models, elicitation techniques.	Knowledge of design process issues including team roles and organization.	Planning requires an understanding of how teams work.	Knowledge needed for formulation of models for process quality.
MAA			Modeling of relationships and analysis for consistency.	Ability to model relationships, structures and interactions for a design model.		
REQ				Input to the design process and used for evaluation of design models.		
DES					Development is essentially design, so managing this needs an awareness of design issues.	Quality issues related to design need an understanding of design goals and fundamentals.

Table 6.2 Rationale for dependencies between knowledge areas.

CG	Summary of Guideline
CG 1	Curriculum designers and instructors must have sufficient relevant knowledge and experience and understand the character of software engineering.
CG 3	Curriculum designers must strike an appropriate balance between material coverage and the flexibility to allow for innovation.
CG 11	The underlying and enduring principles of software engineering should be emphasized, rather than details of the latest or specific tools.
CG 13	Material taught in a software engineering program should, where possible, be grounded in (a) sound empirical research and mathematical or scientific theory or (b) widely accepted good practice.
CG 16	Software process should be central to the curriculum organization and to students' understanding of software engineering practice.
CG 17	To ensure that students embrace certain important ideas, care must be taken to motivate students by using interesting, concrete, and convincing examples.
CG 19	Important efficiencies and synergies can be achieved by designing curricula so that several types of knowledge are learned at the same time.

Quality Issues

Although design models for curricula, like those for software and other artifacts, may not be right or wrong, there are some rather general criteria that might usefully be considered when reviewing a curriculum model:

- Avoid duplication. Although a topic or knowledge unit might well be covered more than once, especially when expanding detail or addressing more advanced

aspects in later courses, there is a need to ensure that teaching about particular issues is not duplicated unnecessarily.

- Ensure that crosscutting issues get appropriate coverage and that ideas are developed systematically. Although such issues are generally best addressed across multiple courses, to emphasize their wider significance, it is necessary to ensure that the key ideas are gradually expanded and to keep track of their development.
- Provide iteration of development experience so that students can reflect, learn, and revise their ideas as appropriate.
- Ensure that there is clear justification for including any material not directly related to software engineering. This is likely to arise when a degree program is part of a portfolio of programs. For example, while the SEEK identifies some branches of mathematics, such as statistics, it does not include calculus because this is rarely used in software engineering, and then it is used only when needed for a particular application domain. There may well be good reasons for including a calculus course (such as giving students the option to change their choice of program at the end of the first year), but this should then be seen as part of a general requirement and not presented as part of the software engineering program.

CG	Summary of Guideline
CG 4	Many software engineering concepts, principles, and issues should be taught as recurring themes throughout the curriculum to help students develop a software engineering mindset.
CG 5	Learning certain software engineering topics requires maturity, so these topics should be taught toward the end of the curriculum, while other material should be taught earlier to facilitate gaining that maturity.
CG 8	Students should be trained in certain personal skills that transcend the subject matter.

6.2 The Capstone Project

A capstone student project is regarded as being an essential element of a software engineering degree program. Such a project provides students with the opportunity to undertake a significant software engineering task, deepening their understanding of many of the knowledge areas forming the SEEK, and with a significant experience at the “a” (application) level of the Bloom taxonomy of learning.

Key characteristics of such a project should include the following:

- The project should span a full academic year, giving students adequate time to reflect upon experiences and retry solutions as appropriate.
- Where possible, this should preferably be undertaken as a group project. If such factors as assessment make this difficult, it is essential that there should be a separate group project of substantial size.

- A project should have some form of implementation as its end deliverable so that the students can experience a wide set of software development activities and adequately evaluate these experiences. Theory-based projects such as the development of formal specifications are therefore inappropriate for this role.
- Evaluation of project outcomes should go beyond concept implementation ("we built it and it worked" [Glass et al., 2004]), using walkthroughs, interviews, or simple experiments to assess the effectiveness and limitations of the deliverables.

Where possible, a project should have a "customer" other than the supervisor so that the student gains fuller experience with product development life-cycle activities.

Assessment of a capstone project should consider how effectively software engineering practices and processes have been employed, including the quality of student reflection on the experience, and not be based only on the delivery of a working system.

CG	Summary of Guideline
CG 14	The curriculum should have a significant real-world basis.

6.3 Patterns for Delivery

At a detailed level, program delivery will depend on many factors, including the need to share with other programs and the availability of staff with relevant knowledge and experience. Two general strategies, or patterns, however, are used in a range of programs. The first strategy begins by addressing software engineering issues (SE-first), while the second starts with computer science in the first year and then introduces software engineering issues later (CS-first). There is no clear evidence to suggest that one of these is necessarily better than the other. Use of a CS-first approach is more common for pragmatic reasons (such as course sharing), although some would advocate that an SE-first approach is better in terms of developing a deeper understanding of software engineering.

An SE-first approach can help to:

- Encourage a student to think as a software engineer from the start, focusing on requirements, design, and verification as well as coding, in preparation for the application of these practices to the development of larger systems.
- Discourage the development of a code-and-fix mentality that may later be difficult to discard. (This can also be achieved with CS-first, but it needs to be explicitly incorporated into that approach.)
- Encourage students to associate themselves with the discipline from the start.

Equally, a CS-first approach has the following benefits:

- It is possible to establish a sound level of programming expertise, thus providing a good basis for understanding software engineering concepts. Programming is a fundamental skill, and a certain level of programming proficiency can facilitate understanding of more abstract software engineering practices. Early exposure

and repeated practice can help to build this proficiency, while appropriate supervision and feedback can minimize the adoption of bad habits.

- Because this is the learning pattern assumed by many textbooks, there is less need to develop dedicated material.
- Software engineering specialists may be in short supply, so the CS-first pattern allows these specialists to focus on teaching later, more specialized, courses.

CG	Summary of Guideline
CG 7	Software engineering must be taught in ways that recognize it is both a computing and an engineering discipline.
CG 10	Software engineering problem solving should be taught as having multiple dimensions.
CG 12	The curriculum must be taught so that students gain experience using appropriate and up-to-date tools, even though tool details are not the focus of the learning.
CG 15	Ethical, legal, and economic concerns and the notion of what it means to be a professional should be raised frequently.
CG 18	Software engineering education needs to move beyond the lecture format and to consider a variety of teaching and learning approaches.

Chapter 7: Adaptation to Alternative Environments

Software engineering curricula do not exist in isolation. They are found in institutions that have differing environments, goals, and practices. Software engineering curricula must be deliverable in a variety of fashions, as part of many different types of institutions.

There are two main categories of "alternate" environments that will be discussed in this section. The first is alternate teaching environments that use nonstandard delivery methods. The second is alternate university organizational models that differ in some significant fashion from the traditional university.

7.1 Alternate Teaching Environments

As higher education has become more universal, the standard teaching environment has tended toward an instructor in the front of a classroom. Although some institutions still retain limited aspects of a tutor-student relationship, the dominant delivery method in most higher education today is classroom-type instruction. The instructor presents material to a class using lecture or lecture/discussion presentation techniques. The lectures may be augmented by appropriate laboratory work. Class sizes range from fewer than 10 to more than 500. Recently, there has been a lot of interest in massive open online courses (MOOCs) that enroll several thousand students each time they are offered. Many MOOCs use the standard lecture format to present new material.

Instruction in the computing disciplines has been notable because of the large amount of experimentation with delivery methods**.** This may be the result of the instructors' familiarity with the capabilities of emerging technologies**.** It may also be the result of the youthfulness of the computing disciplines**.** Regardless of the cause, there are numerous papers in the *SIGCSE Bulletin*, the proceedings of the CSEE&T (Conference on Software Engineering Education & Training), the proceedings of the FIE (Frontiers in Education) conferences, and similar forums that recount significant modifications to the conventional lecture- and lecture/discussion-based classrooms. Examples include all laboratory instruction, the use of electronic whiteboards and tablet computers, problem-based learning, role-playing, activity-based learning, and various studio approaches that integrate laboratory, lecture, and discussion. As mentioned elsewhere in this report, it is imperative that experimentation and exploration be a part of any software engineering curriculum. Necessary curriculum changes are difficult to implement in an environment that does not support experimentation and exploration. A software engineering curriculum will rapidly become out of date unless there is a conscious effort to implement regular change.

If recorded lectures are available for presenting new material, then class time may be used to engage in problem solving and other exercises. This style is sometimes referred to as "flipping the classroom"; students are expected to view the lectures on their own and then come to class prepared to engage in exercises. MOOCs might provide a source for prerecorded lectures.

Much recent curricular experimentation has focused on *distance learning*. The term is not well defined. It can apply to situations where students are in different physical locations but still attend the same scheduled class. This style of learning is often referred to as "synchronous distance learning." Distance learning may also refer to situations where students are in different physical locations but there is no scheduled class time. This style is often referred to as "asynchronous learning." It is important to distinguish between these two cases. It is also important to recognize other cases as well, such as situations where students cannot attend regularly scheduled classes.

Synchronous Learning at Different Physical Locations

Instructing students at different physical locations is a problem that has several solutions. Audio and video links have been used for many years, and broadband Internet connections are now less costly and more accessible. Instructor-student interaction is possible after all involved have learned how to manage the technology without confusion. Two-way video makes such interaction almost as natural as the interaction in a self-contained classroom. Online databases of problems and examples can be used to further support this type of instruction. Web resources, email, and Internet chat can provide a reasonable instructor "office hour" experience. Assignments can be submitted by email or by using a direct Internet connection. The current computing literature and departmental websites contain numerous descriptions of distance learning techniques.

It should be noted that a complete solution to the problem of delivering courses to students in different locations is not a trivial matter, and any solution will require significant planning and appropriate additional support. Some may argue that there is no need to make special provisions for added time and support costs when one merely increases the size of an existing class by adding some "distance" students. Experience indicates that this is always a poor idea.

Students in software engineering programs need to have experience working in teams. Students who are geographically isolated need to be accommodated in some fashion. It is unreasonable to expect that a geographically separated team will be able to do all of its work using email, chat, blogs, and newsgroups; these teams need additional monitoring and support. Videoconferencing and teleconferencing should be considered. Instructors may also want to schedule some meetings with teams, at least via teleconference or videoconference. Beginning students require significantly more monitoring than advanced students because of their lack of experience with geographically separated teams.

One other problem with geographically diverse students is the evaluation of student performance. Appropriate responsible parties will need to be found to proctor examinations and to verify the identities of examinees. Care should be taken to ensure the proper evaluation of student performance. Placing too much reliance on one method (such as written examinations) may make evaluations unreliable.

Asynchronous Learning

Some institutions have a history of providing instruction to "mature" students who are employed in full-time jobs. Because of their work obligations, employed students are often unable to attend regular class meetings. Video recorded lectures, copies of class notes, and electronic copies of class presentations are all useful tools in these situations. A course website, class newsgroup, and class distribution list can provide further support.

Instruction does not necessarily require scheduled class meetings. Self-scheduled and self-paced classes have been used at many institutions. Web-based classes have also been designed. Commercial and open source software has been developed to support many aspects of self-paced and Web-based courses. However, experience shows that the development of self-paced and Web-based instructional materials is expensive and time consuming.

Almost all MOOCs are completely Web-based, providing recorded lectures and self-paced exercises. Some MOOCs also employ automatic grading technology. Although asynchronous learning provides flexibility in scheduling learning activities, most courses still expect students to complete assignments according to a weekly schedule. This helps encourage discipline and makes it possible for students to join study groups with other students enrolled in their courses.

Students who do not have scheduled classroom instruction will still need team activities and experiences. Many of the comments here about geographically diverse teams will also apply to these students as well. An additional problem is created when students are learning at wildly different rates. Because different students will cover content at different times, it is not feasible to have content instruction and projects integrated in the same unit. Self-paced project courses are another serious problem. It is difficult to coordinate team activities when different team members are working at different paces.

7.2 Issues Related to Alternate Institutional Models

Articulation Problems

Articulation problems arise when students have taken one set of courses at one institution or in one program and need to apply these to meet the requirements of a different institution and/or program.

If software engineering curricula existed in isolation, there would be no articulation problems, but this is rarely the case. Software engineering programs are offered by universities with multiple colleges, schools, divisions, departments, and programs as well as by universities that cooperate and compete with one another. Some secondary schools offer university-level instruction, and students expect to receive appropriate credit and placement. Satisfactory completion of a curriculum must be certified when the student has taken classes in different areas of the university as well as at other institutions. Software engineering programs must be designed and managed to minimize articulation problems. This means that the internal and external environment at an institution must be considered when designing a curriculum.

Coordination with Other University Curricula

Many of the core classes in a software engineering curriculum may be shared with programs in related disciplines. An introductory computer science course could be required for the curricula in computer science, computer engineering, and software engineering. Certain architecture courses might be part of curricula in computer science, computer engineering, software engineering, and electrical engineering. Mathematics courses could be required by programs in mathematics, computer science, software engineering, and computer engineering. A project management course may be common to programs in software engineering and management information systems. Upper-level software engineering courses could be taken as part of computer science or computer engineering programs. In most universities, there will be pressure to have courses do "double duty" whenever possible.

Courses that are a part of more than one curriculum must be carefully designed. There is great pressure to include everything of significance to all of the relevant disciplines. This pressure must be resisted because it is impossible to satisfy everyone's desires. Courses that serve two masters will inevitably have to omit topics that would be present were it not for the other master. Curriculum implementers must recognize that perfection is impossible and impractical. The minor content loss when courses are designed to be part of several curricula is more than compensated for by the experience of interacting with students with other ideas and background. Indeed, a case can be made that such experiences are so important in a software engineering curriculum that special efforts should be made to create courses common to several curricula.

Cooperation with Other Institutions

In today's world, students complete their university education via a variety of pathways. Many students attend just one institution, but there are substantial numbers who attend more than one. For a variety of reasons, many students begin their baccalaureate degree program at one institution and complete it at another. In so doing, students may change their career goals or declare new majors; may move from a liberal arts program to an engineering or scientific program; may satisfy interim program requirements at one institution; may engage in work-related experiences; or may be coping with financial, geographic, or personal constraints.

Software engineering curricula must be designed so that these students are able to complete the program without undue delay and repetition, through recognition of comparable coursework and aligned programs. It is straightforward to grant credit for previous work (whether from another department, school, college, or university) when the content of the courses being compared is substantially identical; in other cases, significant problems can arise. Although credit should not be granted for a substitute course that does not cover the intended material, a small amount of missing content should not require that a student repeat an entire course. Faculty do not want to see a student's progress unduly delayed because of articulation issues; therefore, the wisest criteria to use when determining transfer and placement credit are whether the student

can reasonably be expected to address any content deficiencies in a timely fashion and to succeed in subsequent courses.

Student interests will best be served when course equivalencies can be identified and addressed in advance via an articulation agreement. Many institutions have formal articulation agreements with institutions from which they routinely receive transfer students. For example, such agreements are frequently found in the United States between baccalaureate-degree granting institutions and the associate-degree granting institutions that send them transfer students. Other examples can be seen in the 3–2 agreements in the United States between liberal arts and engineering institutions, which allow a student to take three years at a liberal arts institution and two years at an engineering institution, receiving both bachelor of arts and bachelor of science degrees.

When formulating articulation agreements and designing curricula, it is important to consider any accreditation requirements that may exist because the degree-granting program will have to demonstrate that all applicable accreditation criteria have been met for transfer students.

The European Credit Transfer System and the Bologna Process are attempts to reduce articulation problems in Europe.

7.3 Programs for Associate-Degree Granting Institutions in the United States and Community Colleges in Canada

In the United States, as many as one-half of baccalaureate graduates initiated their studies in associate-degree granting institutions. For this reason, it is important to outline a software engineering program of study that can be initiated in the two-year college setting, specifically designed for a seamless transfer into an upper-division (years 3 and 4) program. Regardless of their skills upon entry into the two-year college, students must complete the coursework in its entirety with well-defined competency points to ensure success in the subsequent software engineering coursework at the baccalaureate level. For some students, this may require more than two years of study at the associate level. In any case, the goal is the same: to provide a program of study that prepares the student for the upper-level institution.

Recently, the ACM sponsored the development of curriculum guidelines for two-year college programs that would allow transfer into a baccalaureate program in software engineering [http://www.capspace.org/pgm_inventory/programdetail.aspx?pID=40]. These guidelines are a valuable resource for programs that wish to serve this group of transfer students.

Chapter 8: Program Implementation and Assessment

8.1 Curriculum Resources and Infrastructure

Once a curriculum is established, the success of an educational program critically depends on three specific elements: the faculty, the student body, and the infrastructure. Another important element is ongoing industry involvement with the program.

Faculty

A high-quality faculty and staff is perhaps the single most critical element in the success of a program. Faculty resources must be sufficient to teach the program's courses and support the educational activities needed to deliver the curriculum and reach the program's objectives. The teaching and administrative load must allow time for faculty members to engage in scholarly and professional activities, which are particularly critical because of the dynamic nature of computing and software engineering. (See CG 1 in Chapter 5.)

These faculty members need a strong academic background in software and computing, but they must also have sufficient experience in software engineering practice. At this stage in the development of software engineering as an academic discipline, it can be a challenge to recruit faculty members with the desired combination of academic credentials, effective teaching skills, potential for research and scholarship, and practical software engineering experience [Glass 2003].

Software engineering faculty members should be encouraged and supported in their efforts to become and remain current in industrial software engineering practice through applied research, industry internships, consulting, and so forth. Faculty members with backgrounds in specialized areas of computing may need help in broadening their understanding of current software engineering research and practice.

Because software engineering programs generally incorporate significant laboratory and project experiences that go beyond traditional classroom instruction, additional teaching resources are needed to provide adequate supervision of student work. For example, student teams need to meet regularly with faculty or other supervisors to ensure adequate progress and proper application of software engineering practices and processes. This type of laboratory work is considered routine in other engineering disciplines, but the workload requirements it imposes may be less familiar for faculty and administrators with experience in other areas of computing.

Students

Another critical factor in the success of a program is the quality of its student body. Admission standards should be adequate to assure that students are properly prepared for the program. Student advising and progress monitoring processes support student retention and help to ensure that graduates of the program meet the program objectives and desired outcomes. Appropriate metrics, consistent with the institutional mission and

program objectives, must exist to guide students toward completion of the program in a reasonable period of time and to measure the success of the graduates in meeting the program objectives.

Students are a valuable source of information about the effectiveness of the curriculum structure and course delivery. Involving students in professional organizations and other co-curricular activities can enrich a program's learning environment and support program assessment and continuous improvement.

Infrastructure

The program must provide adequate infrastructure and technical support. These include well-equipped laboratories and classrooms, adequate study areas, and laboratory staff capable of providing adequate technical support. Student project teams need adequate facilities for team meetings, inspections and walkthroughs, customer reviews, and effective communication with instructors or other supervisors. The program must also ensure access to sufficient reference and documentation material as well as library resources in software engineering and related computing disciplines.

Maintaining laboratories and a modern suite of applicable software tools can be a daunting task because of the dynamic, accelerating pace of advances in software and hardware technology. Nevertheless, as pointed out earlier in this document, it is essential that students gain experience using appropriate and up-to-date tools.

An academic program in software engineering must have sufficient leadership and staff to provide proper program administration. This should include adequate levels of student advising, support services, and interaction with relevant constituencies such as employers and alumni. The advisory function of the faculty must be recognized by the institution and must be given appropriate administrative support.

There must be sufficient financial resources to support the recruitment, development, and retention of adequate faculty and staff; the maintenance of an appropriate infrastructure; and all necessary program activities.

Industry Participation

An additional critical element in the success of a software engineering program is the involvement and active participation of industry. Industry advisory boards and industry-academic partnerships help maintain curriculum relevance and currency. Such relations can support various activities including programmatic advice from an industry perspective, student and faculty industry internships, integration of industry projects into the curriculum, guest lectures, and visiting faculty positions from industry.

8.2 Assessment and Accreditation Issues

To maintain a quality curriculum, a software engineering program should be assessed regularly. Many feel assessment is best accomplished in conjunction with a recognized

accreditation organization. Curriculum guidance and accreditation standards and criteria are provided by a number of accreditation organizations across a variety of nations and regions [ABET 2014a, ABET 2014b, BCS 2001,CEAB 2002, ECSA 2000, King 1997, IEI 2000, ISA 1999, JABEE 2003]. In some countries, assessment is a carried out by the government under a standard predefined curriculum model or set of curriculum standards and guidelines.

At the time of this report, ABET had accredited 28 undergraduate programs in software engineering, most of them in the United States. Specific criteria for software engineering programs are included in the ABET *Criteria for Accrediting Engineering Programs* [ABET 2014b].

Accreditation typically includes periodic external program review, which assures that programs meet a minimum set of criteria and adhere to an accreditation organization's standards. A popular approach to assessment and accreditation is an outcomes-based approach for which educational objectives and/or student outcomes are established first; then the curriculum, an administrative organization, and the infrastructure needed to meet the objectives and outcomes are put into place.

The assessment should evaluate the program objectives and desired outcomes. as well as the curriculum content and delivery, and it should serve as the primary feedback mechanism for continuous improvement.

8.3 SE in Relation to Other Computing-Related Disciplines

Software engineering has strong associations with other areas of science and technology, especially those related to computing. Although software engineering is clearly identified by an emphasis on design, a distinctive feature of engineering programs, discerning the precise boundaries that separate the computing disciplines is not always easy.

The Computing Curricula series includes a volume entitled "Computing Curricula 2005: The Overview Report" that "provides undergraduate curriculum guidelines for five defined sub-disciplines of computing: Computer Science, Computer Engineering, Information Systems, Information Technology, and Software Engineering." [CC 2005] Readers are encouraged to consult that volume for a good overview of the similarities and differences between computing disciplines.

Chapter 9: References

[ABET 2014a] ABET, *Accreditation Policy and Procedure Manual*, Nov. 2014; http://www.abet.org/appm-2015-2016/.

[ABET 2014b] ABET, *Criteria for Accrediting Engineering Programs*, Nov. 2014; http://www.abet.org/eac-criteria-2015-2016/.

[ACM 1998] ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices, *Software Engineering Code of Ethics and Professional Practice*, version 5.2, Sept. 1998; http://www.acm.org/serving/se/code.htm.

[BCS 2001] British Computer Society, *Guidelines On Course Exemption & Accreditation For Information For Universities And Colleges*, Aug. 2001; http://www1.bcs.org.uk/link.asp?sectionID=1114.

[Beck 2004] K. Beck, *Extreme Programming Explained: Embrace Change*, 2nd ed., Pearson Addison-Wesley, 2004.

[Beecham et al. 2008] S. Beecham et al., "Motivation in Software Engineering: A Systematic Literature Review," *Information & Software Technology*, vol. 50, 2008, pp. 860–878.

[Bloom 1956] B.S. Bloom, ed., *Taxonomy of Educational Objectives: The Classification of Educational Goals: Handbook I, Cognitive Domain*, Longmans, 1956.

[Boehm 1988] B.W. Boehm, "A Spiral Model of Software Development and Enhancement," *Computer*, vol. 21, no. 5, 1988, pp. 61–72.

[Boehm & Turner 2003] B.W. Boehm and R. Turner, *Balancing Agility and Discipline: A Guide for the Perplexed*, Pearson Addison-Wesley, 2003.

[Brooks 1987] F.P. Brooks, Jr. "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer*, vol. 20, no. 4, 1987, pp. 10–19.

[CEAB 2002] Canadian Eng. Accreditation Board, *Accreditation Criteria and Procedures*, Canadian Council of Professional Engineers, 2002; http://www.ccpe.ca/e/files/report_ceab.pdf.

[CS2013] ACM/IEEE CS Joint Task Force on Computing Curricula. "Computer Science Curricula 2013," ACM Press and IEEE CS Press, 2013,; DOI: http://dx.doi.org/10.1145/2534860

[da Silva 2012] F.Q.B. da Silva et al., "An Evidence-Based Model of Distributed Software Development Project Management: Results from a Systematic Mapping Study," *Software: Evolution & Process*, vol. 24, no. 6, pp. 625–642.

[Denning 1992] P.J. Denning, "Educating a New Engineer," *Comm. ACM*, Dec. 1992.

[ECSA 2000] Eng. Council Of South Africa, *Policy on Accreditation of University Bachelors Degrees*, Aug. 2000; http://www.ecsa.co.za/

[Floyd 1984] C. Floyd, "A Systematic Look at Prototyping," *Approaches to Prototyping* Budde R., Kuhlenkamp K., Mathiassen L., and Zullighoven H., eds., Springer Verlag, 1984, pp. 1–18.

[Gladden 1982] G.R. Gladden, "Stop the Life-Cycle, I Want to Get Off," *ACM Software Eng. Notes*, vol. 7, no. 2, 1982, pp. 35–39.

[Glass 2003] R.L. Glass, "A Big Problem in Academic Software Engineering and a Potential Outside-the-Box Solution," *IEEE Software*, vol. 20, no. 4, 2003, pp. 96, 95.

[Glass et al. 2004] R.L. Glass, V. Ramesh, and I. Vessey, "A Analysis of Research in Computing Disciplines," *Comm. ACM*, vol. 47, no. 6, 2004, pp. 89–94.

[Holcombe 2008] M. Holcombe, *Running an Agile Software Development Project*, Wiley, 2008.

[Hughes 2000] R. Hughes, *Practical Software Measurement*, McGraw Hill, 2000.

[IEEE 2001b] ACM/IEEE-Curriculum 2001 Task Force, *Computing Curricula 2001, Computer Science* , Dec. 2001; http://www.computer.org/education/cc2001/final/index.htm.

[IEEE 2004] ACM/IEEE Joint Task Force on Computing Curricula, *Software Engineering 2004 Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering*, Aug. 2004; http://www.computer.org/portal/c/document_library/get_file?p_l_id=2814020&folderId=3111026&name=DLFE-57602.pdf.

[IEEE 2005] ACM/IEEE Joint Task Force on Computing Curricula, *Computing Curricula 2005 The Overview Report*, Sept. 2005; http://www.computer.org/portal/c/document_library/get_file?p_l_id=2814020&folderId=3111026&name=DLFE-57601.pdf.

[IEEE 2010] ISO/IEC/IEEE 24765:2010 Systems and Software Engineering—Vocabulary.

[IEI 2000] The Institution of Engineers of Ireland, *Accreditation of Engineering Degrees*, May 2000; http://www.iei.ie/Accred/accofeng.pdf.

[ISA 1999] Institution of Engineers, Australia, *Manual For The Accreditation Of Professional Engineering Programs*, Oct. 1999; http://www.ieaust.org.au/membership/res/downloads/AccredManual.pdf.

[JABEE 2003] Japan Accreditation Board for Engineering, *Criteria for Accrediting Japanese Engineering Education Programs 2002-2003*; http://www.jabee.org/english/OpenHomePage/e_criteria&procedures.htm.

[King 1997] W.K. King and G. Engel, *Report on the International Workshop on Computer Science and Engineering Accreditation*, , IEEE CS, 1997.

[Leffingwell 2011] D. Leffingwell, *Agile Software Requirements*, Pearson Addison-Wesley, 2011.

[McCracken & Jackson 1982] D.D. McCracken and M.A. Jackson, "Life-Cycle Concept Considered Harmful," *ACM Software Eng. Notes*, vol. 7, no. 2, 1982, pp. 29–32.

[NACE 2013] Nat'l Assoc. of Colleges and Employers. *Job Outlook 2013*; http://www.naceweb.org/.

[Naur 1969] P. Naur and B. Randell, eds., *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee*, (7–11 October 1968), Scientific Affairs Division, NATO, 1969.

[OECD 2010] *OECD Information Technology Outlook*, www.oecd.org.

[Peters & Tripp 1976] L.J. Peters and L.L. Tripp, "Is Software Design 'Wicked','" *Datamation*, vol. 22, no. 5, 1976, p. 127.

[Pfleeger 2005] S.L. Pfleeger, "Soup or Art? The Role of Evidential Force in Empirical Software Engineering," *IEEE Software*, vol. 22, no. 1, 2005, pp. 66–73.

[Rittel & Webber 1984] H.J. Rittel and M.M. Webber, "Planning Problems Are Wicked Problems," "*Developments in Design Methodology*", N. Cross, ed., 1984, Wiley, pp. 135–144.

[Rogers et al. 2011] Y. Rogers, H. Sharp H., and J. Preece, *Interaction Design: Beyond Human-Computer Interaction*, 3rd ed., Wiley, 2011.

[Schwaber 2004] K. Schwaber, *Agile Project Management with Scrum*, Microsoft Press, 2004.

[Smite et al. 2010] D. Smite et al., "Empirical Evidence in Global Software Engineering: A Systematic Review, *Empirical Software Eng*., vol. 15, 2011, pp. 91–118.

[SWEBOK 2014] P. Bourque and R.E. Fairley, eds., *Guide to the Software Engineering Body of Knowledge, Version 3.0*, IEEE Computer Society, 2014; www.swebok.org.

[Tucker et al. 2011] A. Tucker, R. Morelli, and C. de Silva, *Software Development: An Open Source Approach*, Chapman & Hall, 2011.

[Weinberg 1971] G.M. Weinberg, *The Psychology of Computer Programming*, Van Nostrand Reinhold, 1971.

Appendix A. Curriculum Examples

This appendix contains examples of curricula from undergraduate software engineering programs.

A.1. Mississippi State University

Bachelor of Science in Software Engineering
Mississippi State University, Starkville, MS
Sarah Lee, sblee@cse.msstate.edu

http://cse.msstate.edu/academics/understud/

Program Overview
Mississippi State University is a comprehensive, doctoral-degree-granting university with an overall enrolment a little over 20,000. The Bagley College of Engineering at MSU is a professional college whose purposes are to provide both undergraduate and graduate education, to conduct basic and applied research, and to engage in extension and public service activities. The Department of Computer Science and Engineering offers two majors: computer science and software engineering. Both programs are ABET accredited. Each year an average of 12.7 students earn degrees in software engineering and about 32 students earn degrees in computer science.

Objectives and Expected Outcomes of Program
The software engineering program prepares graduates for a variety of careers in the information technology domain as well as for graduate study in closely related disciplines. Within a few years after graduation, graduates are expected to:

- Demonstrate an understanding of engineering principles and an ability to solve unstructured engineering problems through the successful entrance into and advancement in the software engineering profession.
- Demonstrate an appreciation for lifelong learning and for the value of continuing professional development through participation in graduate education, professional education or continuing education opportunities, attainment of professional licensure, or membership in professional societies.
- Demonstrate an understanding of professional and ethical responsibilities to the profession, society and the environment incumbent on an engineering professional.
- Successfully interact with others of different backgrounds, educations, and cultures.
- Demonstrate effective communication skills in their profession.

The software engineering program enables students to attain, by the time of graduation:

- an ability to apply knowledge of mathematics, science, and engineering
- an ability to design and conduct experiments, as well as to analyze and interpret data

- an ability to design a system, component, or process to meet desired needs within realistic constraints such as economic, environmental, social, political, ethical, health and safety, manufacturability, and sustainability
- an ability to function on multidisciplinary teams
- an ability to identify, formulate, and solve engineering problems
- an understanding of professional and ethical responsibility
- an ability to communicate effectively
- the broad education necessary to understand the impact of engineering solutions in a global, economic, environmental, and societal context
- a recognition of the need for, and an ability to engage in life-long learning
- a knowledge of contemporary issues
- an ability to use the techniques, skills, and modern engineering tools necessary for engineering practice.

Example Study Plan(s)

Freshman Year

First Semester		Second Semester	
CSE 1002 Intro to CSE	2	CSE 1384 Inter Computer Programming	4
MA 1713 Calculus I	3	MA 1723 Calculus II	3
CH 1213 Fundamentals of Chemistry	3	PH 2213 Physics I	3
CH 1211 Investigations in Chemistry	1	EN 1113 English Comp. II	3
EN 1103 English Comp. I	3	CO 1003 Public Speaking	3
CSE 1284 Intro Computer Programming	4		
Total Credit Hours	16	Total Credit Hours	16

Sophomore Year

First Semester		Second Semester	
CSE 2383 Data Struc & Anal.of Algorithms	3	CSE 2813 Discrete Structures	3
ECE 3714 Digital Devices & Logic Design	4	CSE 3324 Distributed Client/Server Prog.	4
MA 2733 Calculus III	3	ECE 3724 Microprocessors I	4
PH 2223 Physics II	3	MA 4th semester Math class*	3
Fine Arts Elective	3	IE 4613 Engineering Statistics I	3
Total Credit Hours	16	Total Credit Hours	17

Junior Year

First Semester		Second Semester	
CSE 4503 Database Management Sys. .	3	Free Elective	3
CSE 4214 Intro to Software Engineering	4	Technical Elective**	3
Social Science Elective	3	CSE 4833 Intro. To Analysis of Algo.	3
CSE 4733 Operating Systems I	3	CSE 4153 Data Com. & Com. Networks	3

BIO 1134 Biology I 4
Social Science Elective 3

Total Credit Hours 17
Total Credit Hours 15

Senior Year

First Semester

CSE 4233 Software Arch. & Design Para. 3
Assurance 3
Security Elective 3
CSE 4223 OR IE 4533 Project Management 3
CSE 3213 Soft. Engr. Senior Project I 3
II 3
Humanities Elective 3
CSE 3981 Social & Ethical Issues in Comp 1
Total Credit Hours 16

Second Semester

CSE 4283 Soft. Testing & Qual.

Technical Elective** 3
Technical Writing (GE 3513) 3
CSE 3223 Soft. Engr. Senior Project

Humanities Elective 3

Total Credit Hours 15

Total Hours 128

Body of Knowledge Coverage
(Other column represents introductory courses or upper level elective courses)

Reference	Knowledge Unit	3213	3223	3813	4214	4223	4233	4283	Other
CMP	**Computing essentials**								
CMP.cf	Computer science foundations								100%
CMP.ct	Construction technologies				50%				50%
CMP.tl	Construction tools				50%				50%
FND	**Mathematical and Engineering Fundamentals**								
FND.mf	Mathematical foundations			100%					
FND.ef	Engineering foundations for software				50%				50%
FND.ec	Engineering economics for software				100%				
PRF	**Professional Practice**								
PRF.psy	Group dynamics / psychology					100%			
PRF.com	Communications skills (specific to SE)					100%			
PRF.pr	Professionalism					100%			
MAA	**Software Modeling and Analysis**								
MAA.md	Modeling foundations			100%					
MAA.tm	Types of models			100%					
MAA.af	Analysis fundamentals			100%					
REQ	**Requirements analysis and specification**								
REQ.rfd	Requirements fundamentals				100%				
REQ.er	Eliciting requirements	75%			25%				
REQ.rsd	Requirements specification & documentation	75%			25%				
REQ.rv	Requirements validation	75%			25%				

DES	**Software Design**								
DES.con	Design concepts				100%				
DES.str	Design strategies				100%				
DES.ar	Architectural design						100%		
DES.hci	Human-computer interaction design						100%		
DES.dd	Detailed design						100%		
DES.ev	Design evaluation		50%				50%		
VAV	**Software verification and validation**								
VAV.fnd	V&V terminology and foundations							100%	
VAV.rev	Reviews and static analysis							100%	
VAV.tst	Testing							100%	
VAV.par	Problem analysis and reporting							100%	
PRO	**Software Process**								
PRO.con	Process concepts				100%				
PRO.imp	Process implementation				100%				
PRO.pp	Project planning and tracking				25%	75%			
PRO.cm	Software configuration management				100%				
PRO.evo	Evolution processes and activities				50%	50%			
QUA	**Software Quality**								
QUA.cc	Software quality concepts and culture							100%	
QUA.pca	Process assurance							100%	
QUA.pda	Product assurance							100%	
SEC	**Security**								
SEC.sfd	Security fundamentals				25%				75%
SEC.net	Computer and network security				25%				75%
SEC.dev	Developing secure software				25%		75%		

Additional Comments (optional)
Students may earn an Information Assurance Professional certificate by completing a minimum of 15 semester credit hours of approved courses.

Appendix: Information on Individual Courses

CSE 1002 Introduction to CSE
Two hours lecture. Introduction to the computer science and software engineering curricula, profession, and career opportunities. Historical perspective; support role of the department. Ethics, team building, problem solving.

CSE 1284 Introduction to Computer Programming
Prerequisites: MA 1313 College Algebra or equivalent
Three hours lecture. Three hours laboratory. Introductory problem solving and computer programming using object-oriented techniques. Theoretical and practical aspects of programming and problem solving. Designed for CSE, CPE and SE majors.

CSE 1384 Intermediate Computer Programming
Prerequisites: CSE 1284 with a grade of C or better

Three hours lecture. Three hour laboratory. Object-oriented problem solving, design, and programming. Introduction to data structures, algorithm design and complexity. Second course in sequence designed for CSE, CPE and SE majors.

CSE 2383 Data Structures and Analysis of Algorithms
Prerequisites: CSE 1384 and MA 1713 Calculus 1, both with a grade of C or better
Three hours lecture. Non-linear data structures and their associated algorithms. Trees, graphs, hash tables, relational data model, file organization. Advanced software design and development.

CSE 2813 Discrete Structures
Prerequisites: CSE 1284 and MA 1313 College Algebra, both with a grade of C or better

Three hours lecture. Concepts of algorithms, induction, recursion, proofs, topics from logic, set theory, combinatorics, graph theory fundamental to study of computer science.

CSE 3213 Software Engineering Senior Project I
Prerequisites: CSE 4214 with a grade of C or better
Six hours laboratory. Software requirements elicitation and specification, cost estimation, scheduling, development of project management and quality assurance plans, reviews.

CSE 3223 Software Engineering Senior Project II
Prerequisites: CSE 4214 with a grade of C or better
Six hours Laboratory. Team work, software design, construction, implementation of project management and quality assurance plans, and configuration management.

CSE 3324 Distributed Client/Server Programming
Prerequisites: CSE 2383 with a grade of C or better
Three hours lecture. Three hours laboratory. Design of software systems for use in distributed environments. Client/Server models, multithreaded programming, server-side web programming, graphical user interfaces; group projects involving client/server systems.

CSE 3981 Social and Ethical Issues in Computing
Prerequisites: Senior Standing
One hour lecture. Study of major social and ethical issues in computing, including history of computing, impact of computers on society, and the computer professional\’s code of ethics.

CSE 4153 Data Communications and Computer Networks
Prerequisites: CSE 1384 and ECE 3724 Microprocessors, both with a grade of C or better
Three hours lecture. The concepts and practices of data communications and networking to provide the student with an understanding of the hardware and software used for data communications.

CSE 4214 Introduction to Software Engineering
Prerequisites: CSE 2383 with a grade of C or better
Three hours lecture. Two hours laboratory. Introduction to software engineering: planning, requirements, analysis and specification, design; testing; debugging; maintenance; documentation. Alternative design methods, software metrics, software project management, reuse and reengineering.

CSE 4223 Management of Software Projects
Prerequisites: CSE 4214 with a grade of C or better
Three hours lecture. Concepts in software project management functions such as planning, organizing, staffing, directing and control, estimating, scheduling, monitoring, risk management, and use of tools.

CSE 4233 Software Architecture and Design Paradigms
Prerequisites: CSE 4214 with a grade of C or better
Three hours lecture. Topics include software architectures, methodologies, model representations component-based design, patterns, frameworks, CASE-based designs, and case studies.

CSE 4283 Software Testing and Quality Assurance
Prerequisites: CSE 4214 with a grade of C or better
Three hours lecture. Topics include methods of testing, verification and validation, quality assurance processes and techniques, methods and types of testing, and ISO 9000/SEI CMM process evaluation.

CSE 4503 Database Management Systems
Prerequisites: CSE 2383 and CSE 2813, both with a grade of C or better
Three hours lecture. Modern database models; basic database management concepts; query languages; database design through normalization; advanced database models; extensive database development experience in a team environment.

CSE 4733 Operating Systems I
Prerequisites: CSE 2383 and ECE 3724 Microprocessors, both with a grade of C or better
Three hours lecture. Historical development of operating systems to control complex computing systems; process management, communication, scheduling techniques; file system concepts and operation; data communication, distributed process management.

CSE 4833 Introduction to Analysis of Algorithms
Prerequisites: CSE 2383, CSE 2813, and MA 2733 Calculus 3 all with a grade of C or better
Three hours lecture. Study of complexity of algorithms and algorithm design. Tools for analyzing efficiency; design of algorithms, including recurrence, divide-and-conquer, dynamic programming, and greedy algorithms.

A.2. Rose-Hulman Institute of Technology

Bachelor of Science in Software Engineering
Rose-Hulman Institute of Technology, Terre Haute, IN
Mark Ardis, mark.ardis@stevens.edu, Steve Chenoweth, chenowet@rose-hulman.edu

http://www.rose-hulman.edu/course-catalog/course-catalog-2013-2014/programs-of-study/software-engineering.aspx

Program Overview
Rose-Hulman Institute of Technology is a science and engineering school with about 2100 undergraduate and 100 graduate students. The Department of Computer Science and Software Engineering offers two majors: computer science and software engineering. Both programs are ABET accredited. Each year about 35 students earn degrees in software engineering and about 50 students earn degrees in computer science.

Objectives and Expected Outcomes of Program
The software engineering program prepares its graduates for many types of careers in the computing industry as well as for graduate study in software engineering and in closely related disciplines. Within a few years after completing the software engineering degree program, our graduates will:

- Advance beyond their entry-level position to more responsible roles, or progress towards completion of advanced degree(s).
- Continue to keep pace with advancements in their disciplines, and develop professionally in response to changes in roles and responsibilities.
- Demonstrate that they can collaborate professionally within or outside of their disciplines at local, regional, national, or international levels.
- Contribute to the body of computing products, services, or knowledge.

By the time students graduate with a Software Engineering degree from Rose-Hulman, they will be able to:

- Apply software engineering theory, principles, tools and processes, as well as the theory and principles of computer science and mathematics, to the development and maintenance of complex, scalable software systems.
- Design and experiment with software prototypes
- Select and use software metrics
- Participate productively on software project teams involving students from a variety of disciplines
- Communicate effectively through oral and written reports, and software documentation
- Elicit, analyze and specify software requirements through a productive working relationship with project stakeholders

- Evaluate the business and impact of potential solutions to software engineering problems in a global society, using their knowledge of contemporary issues
- Explain the impact of globalization on computing and software engineering
- Interact professionally with colleagues or clients located abroad and overcome challenges that arise from geographic distance, cultural differences, and multiple languages in the context of computing and software engineering
- Apply appropriate codes of ethics and professional conduct to the solution of software engineering problems
- Identify resources for determining legal and ethical practices in other countries as they apply to computing and software engineering
- Recognize the need for, and engage in, lifelong learning
- Demonstrate software engineering application domain knowledge

Example Study Plan

Rose is on the quarter system, with 3 academic terms per year.

Freshman Year								
Fall Term		**Cr**	**Winter Term**		**Cr**	**Spring Term**		**Cr**
CSSE 120	Introduction to Software Development	4	CSSE 220	Object-Oriented Software Development	4	CSSE 132	Introduction to Computer Systems Design	4
MA 111	Calculus I	5	MA 112	Calculus II	5	MA 113	Calculus III	5
PH 111	Physics I	4	PH 112	Physics II	4	HSS	Elective	4
RH 111	Rhetoric & Composition	4	HSS	Elective	4	Science	Elective	4
CLSK 100	College and Life Skills	1						
Sophomore Year								
Fall Term		**Cr**	**Winter Term**		**Cr**	**Spring Term**		**Cr**
CHEM 111	General Chemistry I	4	CSSE 230	Data Structures and Algorithm Analysis	4	CSSE 304	Programming Language Concepts	4
CSSE 232	Computer Architecture I	4	CSSE 333	Database Systems	4	CSSE 376	Software Quality Assurance	4
MA 212	Matrix Alg & Syst of Differtl Equa	4	MA 375	Discrete & Comb Algebra II	4	MA	Elective	4
MA 275	Discrete & Combinatorial Algebra I	4	Domain	Domain track course	4	RH 330	Technical and Professional Communication	4
Junior Year								
Fall Term		Cr	**Winter Term**		**Cr**	**Spring Term**		**Cr**
CSSE 371	Software Requirements Engineering	4	CSSE 332	Operating Systems	4	CSSE 373	Formal Methods in Specification & Design	4
CSSE 372	Software Project Management	4	CSSE 374	Software Design	4	CSSE 375	Software Construction and Evolution	4

MA 381	Introduction to Probability with Statistical Applications	4	HSS	Elective	4	HSS	Elective	4
Domain	Domain track course	4	Domain	Domain track course	4	Dom/Free	Domain track course or free elective	4
Senior Year								
Fall Term		**Cr**	**Winter Term**		**Cr**	**Spring Term**		**Cr**
CSSE 477	Software Architecture	4	CSSE 498	Senior Project II	4	CSSE 499	Senior Project III	4
CSSE 497	Senior Project I	4	CSSE	Elective	4	HSS	Elective	4
HSS	Elective	4	HSS	Elective	4	Free	Elective	4
Dom/Free	Domain track course or free elective	4	Free	Elective	4			

Course prefix explanations:

CLSK	College and Life Skills
CSSE	Computer Science and Software Engineering
Dom	Elective in chosen domain track
HSS	Humanities
MA	Math
PH	Physics
RH	Rhetoric

Body of Knowledge Coverage

The "Other" column covers introductory computer science courses in the program. These are generalizations – much more detail for some of the courses is found in their individual course descriptions.

Reference	Knowledge Unit	371	372	373	374	375	376	477	Other
CMP	**Computing essentials**								
CMP.cf	Computer science foundations								100%
CMP.ct	Construction technologies					50%			50%
CMP.tl	Construction tools					50%			50%
	Mathematical and Engineering								
FND	**Fundamentals**								
FND.mf	Mathematical foundations								100%
FND.ef	Engineering foundations for software								50%
FND.ec	Engineering economics for software		100%						
PRF	**Professional Practice**								
PRF.psy	Group dynamics / psychology		100%						
PRF.com	Communications skills (specific to SE)								
PRF.pr	Professionalism		100%						
MAA	**Software Modeling and Analysis**								
MAA.md	Modeling foundations			100%					
MAA.tm	Types of models			100%					

MAA.af	Analysis fundamentals			100%				
REQ	**Requirements analysis and specification**							
REQ.rfd	Requirements fundamentals	100%						
REQ.er	Eliciting requirements	100%						
REQ.rsd	Requirements specification & documentation	100%						
REQ.rv	Requirements validation	100%						
DES	**Software Design**							
DES.con	Design concepts				100%			
DES.str	Design strategies				100%			
DES.ar	Architectural design						100%	
DES.hci	Human-computer interaction design				100%			
DES.dd	Detailed design				100%			
DES.ev	Design evaluation				50%		50%	
VAV	**Software verification and validation**							
VAV.fnd	V&V terminology and foundations					100%		
VAV.rev	Reviews and static analysis					100%		
VAV.tst	Testing					100%		
VAV.par	Problem analysis and reporting					100%		
PRO	**Software Process**							
PRO.con	Process concepts		100%					
PRO.imp	Process implementation		100%					
PRO.pp	Project planning and tracking		100%					
PRO.cm	Software configuration management		50%		50%			
PRO.evo	Evolution processes and activities				100%			
QUA	**Software Quality**							
QUA.cc	Software quality concepts and culture					100%		
QUA.pca	Process assurance					100%		
QUA.pda	Product assurance					100%		
SEC	**Security**							
SEC.sfd	Security fundamentals							100%
SEC.net	Computer and network security							50%
SEC.dev	Developing secure software				50%			

Additional Comments

Each student completes a sequence of courses in an application domain. These are typically 4 to 6 courses in an area of interest to the student. These domain tracks need to be approved by the department. Most other majors, or minors, also can play this role for a software engineering major.

Appendix: Information on Individual Courses

CSSE 120 Introduction to Software Development

An introduction to procedural and object-oriented programming with an emphasis on problem solving. Problems may include visualizing scientific or commercial data, interfacing with external hardware such as robots, or solving numeric problems from a variety of engineering disciplines. Procedural programming concepts covered include data types, variables, control structures, arrays, and data I/O. Object-oriented programming concepts covered include object creation and use, object interaction, and the design of simple classes. Software engineering concepts covered include testing, incremental development, understanding requirements, and teamwork.

CSSE 132 Introduction to Computer Systems Prereq: CSSE 120
Provides students with an understanding of system level issues and their impact on the design and use of computer systems. Examination of both hardware and software layers. Basic computation structures and digital logic. Representation of instructions, integers, floating point numbers and other data types. System requirements, such as resource management, security, communication and synchronization, and their hardware and/or software implementation. Exploration of multiprocessor and distributed systems. Course topics will be explored using a variety of hands-on assignments and projects.

CSSE 220 Object-Oriented Software Development Prereq: CSSE 120
Object-oriented programming concepts, including the use of inheritance, interfaces, polymorphism, abstract data types, and encapsulation to enable software reuse and assist in software maintenance. Recursion, GUIs and event handing. Use of common object-based data structures, including stacks, queues, lists, trees, sets, maps, and hash tables. Space/time efficiency analysis. Testing. Introduction to UML.

CSSE 230 Data Structures and Algorithm Analysis Prereq: CSSE220 or CSSE 221 with a grade of C or better, and MA 112
This course reinforces and extends students' understanding of current practices of producing object-oriented software. Students extend their use of a disciplined design process to include formal analysis of space/time efficiency and formal proofs of correctness. Students gain a deeper understanding of concepts from CSSE 220, including implementations of abstract data types by linear and non-linear data structures. This course introduces the use of randomized algorithms. Students design and implement software individually, in small groups, and in a challenging multi-week team project.

CSSE 232 Computer Architecture I Prereq: CSSE132, or CSSE120 and ECE130
Computer instruction set architecture and implementation. Specific topics include historical perspectives, performance evaluation, computer organization, instruction formats, addressing modes, computer arithmetic, ALU design, floating-point representation, single-cycle and multi-cycle data paths, and processor control. Assembly language programming is used as a means of exploring instruction set architectures. The final project involves the complete design and implementation of a miniscule instruction set processor.

CSSE 333 Database Systems Prereq: MA275 and CSSE230 (or concurrent enrollment in CSSE230)

Relational database systems, with emphasis on entity relationship diagrams for data modeling. Properties and roles of transactions. SQL for data definition and data manipulation. Use of contemporary API's for access to the database. Enterprise examples provided from several application domains. The influence of design on the use of indexes, views, sequences, joins, and triggers. Physical level data structures: B+ trees and RAID. Survey of object databases.

CSSE 371 Software Requirements Engineering Prereq: CSSE230, RH330, and Junior standing
Basic concepts and principles of software requirements engineering, its tools and techniques, and methods for modeling software systems. Topics include requirements elicitation, prototyping, functional and non-functional requirements, object-oriented techniques, and requirements tracking.

CSSE 372 Software Project Management Co-requ: CSSE371
Major issues and techniques of project management. Project evaluation and selection, scope management, team building, stakeholder management, risk assessment, scheduling, quality, rework, negotiation, and conflict management. Professional issues including career planning, lifelong learning, software engineering ethics, and the licensing and certification of software professionals.

CSSE 373 Formal Methods in Specification and Design Prereq: CSSE230 and MA275
Introduction to the use of mathematical models of software systems for their specification and validation. Topics include finite state machine models, models of concurrent systems, verification of models, and limitations of these techniques.

CSSE 374 Software Design Prereq: CSSE371
Introduction to the design of complete software systems, building on components and patterns. Topics include architectural principles and alternatives, design documentation, and relationships between levels of abstraction.

CSSE 375 Software Construction and Evolution Prereq: CSSE374
Issues, methods and techniques associated with constructing software. Topics include detailed design methods and notations, implementation tools, coding standards and styles, peer review techniques, and maintenance issues.

CSSE 376 Software Quality Assurance Prereq: CSSE230
Theory and practice of determining whether a product conforms to its specification and intended use. Topics include software quality assurance methods, test plans and strategies, unit level and system level testing, software reliability, peer review methods, and configuration control responsibilities in quality assurance.

CSSE 477 Software Architecture Prereq: CSSE374 or consent of instructor
This is a second course in the architecture and design of complete software systems, building on components and patterns. Topics include architectural principles and alternatives, design documentation, relationships between levels of abstraction, theory

and practice of human interface design, creating systems which can evolve, choosing software sources and strategies, prototyping and documenting designs, and employing patterns for reuse. How to design systems which a team of developers can implement, and which will be successful in the real world.

CSSE 497 Senior Project I Prerequisite: CSSE371
CSSE 498 Senior Project II Prerequisite: CSSE 374 and CSSE497
CSSE 499 Senior Project III Prerequisite: CSSE498
Group software engineering project requiring completion of a software system for an approved client. Tasks include project planning, risk analysis, use of standards, prototyping, configuration management, quality assurance, project reviews and reports, team management and organization, copyright, liability, and handling project failure.

Appendix B. Course Examples

This appendix contains examples of software engineering courses from undergraduate software engineering programs.

B.1. Management of Software Projects (MSU)

CSE 4223 Management of Software Projects
Mississippi State University, Starkville MS
Sarah B. Lee
sblee@cse.msstate.edu
http://www.cse.msstate.edu/academics/understud/courses.php

Catalog description
Three hours lecture. Concepts in software project management functions such as planning, organizing, staffing, directing and control, estimating, scheduling, monitoring, risk management, and use of tools.

Expected Outcomes

- **The student should be able to describe alternative software project life cycle models and select the correct model for a given software project scenario.**
- **The student is able to plan tasks, plan task dependencies, estimate effort, and estimate other needed resources.**
- **The student is able to recognize and categorize risks, intellectual property, and legal issues of software projects.**
- **The student is able to organize project personnel and has knowledge of personnel management issues.**

Where does the course fit in your curriculum?
This is a required course taken by all undergraduate software engineering majors. Students typically take the course in the first semester of their fourth year. The course is also open to graduate students in computer science. CSE 4214 Introduction to Software Engineering is a pre-requisite. About 25-30 students take the course each time it is offered.

What is covered in the course?
Life cycle models
Project Planning
Organization Planning
Risk Management
Leadership and Managing personnel

What is the format of the course?
The lecture-based course meets 3 hours per week during a 16 week semester.

How are students assessed?
Students have homework assignments that provide them with hands-on experience with software project planning. Additional homework assignments involve summary of reading assignments dealing with leadership of software development projects. Three exams are given throughout the semester.

Course textbooks and materials
Historically the textbook has been:
Futrell et al., *Quality Software Project Management*, Prentice Hall, 2002
Additional readings are used for some topics. For this coming year, the following book will be used: Tom DeMarco, *The Deadline: A Novel About Project Management*

Pedagogical Advice
Class time is sometimes used for open discussion of readings on leadership. Also use role play to demonstrate management styles in brief scenarios.

Body of Knowledge coverage

Reference	Knowledge Unit	Class Hours
PRF	**Professional Practice**	
PRF.psy	Group dynamics / psychology	5
PRF.com	Communications skills (specific to SE)	5
PRF.pr	Professionalism	4
PRO	**Software Process**	
PRO.con	Process concepts	
PRO.imp	Process implementation	
PRO.pp	Project planning and tracking	12
PRO.cm	Software configuration management	
PRO.evo	Evolution processes and activities	13

Additional topics (optional)

Other comments
Much emphasis is placed on developing leadership skills and the importance of those in managing software projects.

B.2. Software Requirements Engineering (RHIT)

CSSE 371, Software Requirements Engineering
Rose-Hulman Institute of Technology, Terre Haute, IN, USA
Instructors: Sriram Mohan, Steve Chenoweth, Chandan Rupakheti
Email Addresses: mohan@rose-hulman.edu, chenowet@rose-hulman.edu, rupakhet@rose-hulman.edu
URL for additional information: http://www.rose-hulman.edu/class/csse/csse371/

Catalog description
Basic concepts and principles of software requirements engineering, its tools and techniques, and methods for modeling software systems. Topics include requirements elicitation, prototyping, functional and non-functional requirements, object-oriented techniques, and requirements tracking.

Expected Outcomes

Students that successfully complete the course will be able to:

1. Explain the role of requirements engineering and its process.
2. Formulate a problem statement using standard analysis techniques.
3. Determine stakeholder requirements using multiple standard techniques
4. Produce a specification with functional and non-functional requirements based on the elicited requirements.
5. Decide scope and priorities by negotiating with the client and other stakeholders.
6. Manage requirements.
7. Apply standard quality assurance techniques to ensure that requirements are: verifiable, traceable, measurable, testable, accurate, unambiguous, consistent, and complete.
8. Produce test cases, plans, and procedures that can be used to verify that they have defined, designed and implemented a system that meets the needs of the intended users.
9. Design and Prototype user interfaces to validate requirements.
10. Prepare and conduct usability tests to evaluate the usability, utility and efficiency of the developed user interface.

Where the course fits into our curriculum

Normally taught in:
Fall of junior year for almost all students.

Course Prerequisites:
CSSE 230 (Fundamentals of Software Development III, our data structures course) or equivalent; RH 330 or equivalent (our second technical writing course); and Junior standing. The latter is important, because many students are able to get a summer internship after their sophomore year. That is often effective in increasing their interest

in software engineering practices and, sometimes, in the value of getting good requirements, in particular.

Normally this course follows:
RH330 (see above), and CSSE 376 (software quality assurance).

Normally followed immediately by:
CSSE 374 (Software Design)

Also required for:
CSSE 497-8-9 (3-term Senior Project sequence)

What is covered in the course?

The course nominally is about software requirements, but, equally important, it gives students experiential learning on a large software project. It is a problem-based learning course, with the project being the problem. For realism, it is a new project each year, and it involves unknowns that even the instructor cannot predict. A decision needs to be made regarding how much course material should be emphasized, if it is not going to be an integral part of the project.

The course additionally is intended to be transformative learning (in the sense of Jack Mezirow, *et al*[1]), with abrupt obstacles faced that have not been encountered before by our students, and their doing critical reflection and other practices to grow from these experiences. This type of learning is a core part of a software education and not usually included intentionally in a computer science education. An example of such an experience, in this course, is having a client change their mind about requirements, forcing the student teams to backtrack; while, at the same time, the students must maintain a cordial relationship with the client.

We do not have a separate required course on interaction design, but we believe that all students should be able to apply major concepts in this area. Thus, that major topic is included in this requirements course.

What is the format of the course?

The course is taught as an hour discussion / work activity 3 times a week, and a 3-hour intensive project lab once a week. There is some lecture, but this is not a dominant part of even discussion times. The goal of every class session is for individual students and teams to be able to apply requirements-related and other skills as soon as possible, with growing independence, and to learn from that application.

How are students assessed?

---

[1] See, for example, Jack Mezirow, Edward W. Taylor, and Associates. *Transformative Learning in Practice: Insights from Community, Workplace, and Higher Education.* Jossey-Bass, 2009.

Project - Each student is part of a team project with an external entity or Rose-Hulman faculty or staff member as a client; each team may have a different project, or, alternatively, all teams in a section may have the same client for a large project. In 2013-14, all course sections did a large project, which was divided into teams of about 4 within the whole section of about 20 students. These projects continued from this course, CSSE 371 in the fall term, through the software design course, CSSE 374 in the winter, and, for software engineering majors, the software construction and evolution course, CSSE 375 in the spring.

Journals – As an integral part of the project, students are expected to keep technical journals in which they record both team and individual activities that were a part of their work. These are intended, in particular, to demonstrate that the students did critical thinking as a part of the project problem solving. Thus, the journals are graded on this basis, and cannot simply log what happened without showing a search for root causes, development of creative ideas, reflection on teaming experiences, and how they made personal contributions to the team. Along with other ways to judge individual contributions on teams, these journals can be used as a subjective means to verify the significance of those contributions.

Homework – Up to ten homework assignments performed a various times throughout the term are used to apply and reinforce material from the course lectures and discussions and to guide students as they proceed through the project life cycle. Homework assignments also encompass case studies discussed in class to illustrate the importance of requirements engineering in determining project success and failure.

Examinations – Up to three exams may be used to test the students' knowledge and capabilities regarding the course material.

Quizzes – More than 30 short (5-10 questions) quizzes completed before or during class.

Course Assessment Matrix

Assessment Tool	*Outcome*									
	1	*2*	*3*	*4*	*5*	*6*	*7*	*9*	*10*	*11*
Project	X	X	X	X	X	X	X	X	X	X
Homework		X	X	X		x	x	x	x	X
Examinations	X	X		X				X	X	
Quizzes	x									

Success Criteria Grading for the project will be done over five separate milestones and provides an opportunity to evaluate each tool outcome pair multiple times. The course will be considered fully successful if the following statement holds for every tool-outcome pair selected above.

Among the students who earn proficient grades in the course, the average grade on the portions of the assessment tools that are relevant to the learning outcome is in the proficient range.

Course textbooks and materials

- *Managing Software Requirements: A Use Case Approach*, **Second Edition**, by Dean Leffingwell and Don Widrig
- *Interaction Design: Beyond Human-Computer Interaction*, **Third Edition**, by Jennifer Preece, Yvonne Rogers and Helen Sharp

Both textbooks are required, and discussions and assignments come from them directly.

Pedagogical advice

For a course with a large, integral project, the selection and management of that project is as important as the content taught. Projects spanning the full year, and three different courses, must be predicted to have an appropriate level of difficulty and type of activity required that fairly closely matches all three courses. Clients must be found who are willing to dedicate a larger amount of their time than usually is expected even for a senior design team. And these should not be "comfortable" clients if, say, the requirements elicitation process is to be realistic. The entire section (of about 20 students) working on each large project met weekly or biweekly with their client, to show progress. As an example of the sorts of planning that need to be done, clients normally should be available to meet with the sections representing their project, at one of the times that section is scheduled!

We used SCRUM, with two week sprints. Thus, having course material in the three courses progress at the usual pace did not fit this agile method. Students must get enough initial material and skill on topics like configuration management, system design, and testing that they can begin to deliver a progressive product even during this requirements course.

In teaching software engineering subjects to undergrads who lack significant industry experience, one must be aware constantly of the need to relate the learning to something tangible. Having an ongoing class project is our solution for that. One must be equally on patrol for the problem of teaching to a higher level of experience, if the instructor has that higher level herself. For example, teaching *all* the different process alternatives, before the nascent developers have mastered and felt confidence in one of them. Most requirements books used to make this mistake, and that really made them inappropriate for getting students to use one technique really well.

Body of Knowledge coverage

Note that the "contact hours" listed in the right-hand column are a rather rubbery number. We all see this in senior design courses, because it is self-regulated and projects differ in

the amount of work of each type. In this requirements course, the major project is a similar source of variation. While the course provides more guidance than is true in senior design, the goal is for students to do as much as possible, on teams, on their own. The course meets for 10 weeks, plus a final exam week. So there are 6 hours per week times 10, or 60 "contact hours" total. The 3 hours per week of lab are considered "contact hours," because students are doing instructor-supervised project work during that time. The 60 available hours are shown divided up in the table below.

KA	Topic	Hours
REQ	**Requirements analysis and specification**	30 Total
REQ.rfd	**Requirements fundamentals**	6 total
REQ.rfd.1	Definition of requirements (e.g. product, project, constraints, system boundary, external, internal, etc.)	1
REQ.rfd.2	Requirements process	.5
REQ.rfd.3	Layers/levels of requirements (e.g. needs, goals, user requirements, system requirements, software requirements, etc.)	.5
REQ.rfd.4	Requirements characteristics (e.g. testable, non-ambiguous, consistent, correct, traceable, priority, etc.)	.5
REQ.rfd.5	Analyzing quality (non-functional) requirements (e.g. safety, security, usability, performance, root cause analysis, etc.)	.5
REQ.rfd.6	Software requirements in the context of systems engineering	.5
REQ.rfd.7	Requirements evolution	.5
REQ.rfd.8	Traceability	.5
REQ.rfd.9	Prioritization, trade-off analysis, risk analysis, and impact analysis	.5
REQ.rfd.10	Requirements management (e.g. consistency management, release planning, reuse, etc.)	.5
REQ.rfd.11	Interaction between requirements and architecture	.5
REQ.er	**Eliciting requirements**	10 total
REQ.er.1	Elicitation sources (e.g. stakeholders, domain	1

	experts, operational and organization environments, etc.)	
REQ.er.2	Elicitation techniques (e.g. interviews, questionnaires/surveys, prototypes, use cases, observation, participatory techniques, etc.)	9
REQ.rsd	**Requirements specification & documentation**	10 total
REQ.rsd.1	Requirements documentation basics (e.g. types, audience, structure, quality, attributes, standards, etc.)	3
REQ.rsd.2	Software requirements specification techniques (e.g., plan-driven requirements documentation, decision tables, user stories, behavioral specifications)	7
REQ.rv	**Requirements validation**	4 total
REQ.rv.1	Reviews and inspection	.5
REQ.rv.2	Prototyping to validate requirements	1
REQ.rv.3	Acceptance test design	1
REQ.rv.4	Validating product quality attributes	1
REQ.rv.5	Requirements interaction analysis (e.g. feature interaction)	.5
REQ.rv.6	Formal requirements analysis	0 (this is part of a separate course, CSSE 373, in our curriculum.)
MAA	**Software Modeling and Analysis**	5 Total
MAA.tm	Types of models	1
MAA.tm.2	Behavioral modeling (e.g. use case analysis, activity diagrams, interaction diagrams, state machine diagrams, etc.)	3
MAA.tm.4	Domain modeling (e.g. domain engineering approaches, etc.)	1
PRF	**Professional Practice**	10 Total

PRF.psy.1	Dynamics of working in teams/groups	2
PRF.psy.2	Interacting with stakeholders	1
PRF.psy.3	Dealing with uncertainty and ambiguity	1
PRF.psy.4	Dealing with multicultural environments	1
PRF.com	Communications skills (specific to SE)	1
PRF.com.1	Reading, understanding and summarizing reading (e.g. source code, documentation)	1
PRF.com.2	Writing (assignments, reports, evaluations, justifications, etc.)	1
PRF.com.3	Team and group communication (both oral and written)	1
PRF.com.4	Presentation skills	1
DES	**Software Design**	10 Total
DES.con	Design concepts	1
DES.con.3	Context of design within multiple software development life cycles	.5
DES.con.4	Design principles (information hiding, cohesion and coupling)	.5
DES.con.6	Design trade-offs	.5
DES.str	Design strategies	.5
DES.ar	Architectural design	.5
DES.ar.2	Architectural trade-offs among various attributes	.5
DES.ar.4	Requirements traceability in architecture	.5
DES.hci	Human-computer interaction design	1
DES.hci.1	General HCI design principles	1
DES.hci.2	Use of modes, navigation	.5
DES.hci.7	Human-computer interaction design methods	1
DES.hci.8	Interface modalities (e.g., speech and natural language, audio/video, tactile, etc.)	1
DES.hci.9	Metaphors and conceptual models	.5

DES.hci.10	Psychology of HCI	.5
VAV.tst	Testing	5 Total
VAV.tst.5	Human-based testing (e.g., black box testing, domain knowledge)	5

Additional topics

Students are expected to participate in course improvement. This means getting their feedback, and taking pre and post-course questionnaires regarding their level of understanding of course topics, among other things.

Other comments

Note the fact that this course starts a 3-course project. This means, among other things, that students will end the course not having the clean feeling of completing something delivered to the instructor or to a customer. This tends to alter the types of comments we get from them at the end of the course. Many software students rate the value of their own learning on having been successful in finishing the major class project, and they won't get that finality here.

In trade for that, students get to work on projects which are large enough that the software practices do make a difference in success of the project, something they will lose if the project is small and all the non-coding parts are felt to be trivial.

As a system, we get high school students as inputs and produce people impedance-matched for the software industry as outputs. That's the purpose of an undergraduate software engineering program, and the fact we can come close to this model is its advantage over a traditional computer science program. Our industry has a lot of uncertainty and risk involved in each project. That lack of clarity goes against the expectations of most rising juniors, based on the formulaic CS courses they've taken prior to this requirements course. We believe one role of the course is to get the students to mature in their ability to handle such unevenness. This is the transformative learning side of it. The induced practice, in starting to deal with insecure situations, is an important contribution of this course.

B.3. Software Project Management (RHIT)

CSSE 372, Software Project Management
Rose-Hulman Institute of Technology, Terre Haute, IN, USA
Instructors: Michael Hewner, Steve Chenoweth, Shawn Bohner
Email Addresses: hewner@rose-hulman.edu, chenowet@rose-hulman.edu, bohner@rose-hulman.edu
URL for additional information: http://www.rose-hulman.edu/class/csse/csse372/201410/Schedule/Schedule.htm

Catalog description
Major issues and techniques of project management. Project evaluation and selection, scope management, team building, stakeholder management, risk assessment, scheduling, quality, rework, negotiation, and conflict management. Professional issues including career planning, lifelong learning, software engineering ethics, and the licensing and certification of software professionals.

Expected Outcomes

Students who complete this course should be able to:
1. Explain fundamental elements of Software Project Management
2. Identify and explain contemporary software life cycle processes, activities, and work products
3. Estimate software project effort, cost, and schedule for an intermediate size project
4. Identify, analyze, and manage software project risks
5. Create and maintain a software project schedule
6. Create a plan for an intermediate size software project and manage to the plan as project evolves
7. Formulate software project teams in terms of roles and responsibilities
8. Plan, organize and conduct effective project meetings

Where the course fits into our curriculum

Normally taught in:
Fall of junior year for almost all students.

Course Prerequisites:
CSSE 371 (Software Requirements) is a co-requisite. The prerequisites for this course are CSSE 230 (Fundamentals of Software Development III, our data structures course) or equivalent; RH 330 or equivalent (our second technical writing course); and Junior standing. The latter is important, because many students are able to get a summer internship after their sophomore year. That is often effective in increasing their interest in software engineering practices and, sometimes, in the value of getting experience working with project managers, in particular.

Normally this course follows:

RH330 (see above), and CSSE 376 (software quality assurance).

Normally followed immediately by:
CSSE 374 (Software Design)

What is covered in the course?

The course is about project management, which is likely a position that about a third of the Rose-Hulman software engineering graduates will actually hold, though not as their first job. The rest will almost surely have a project manager, or a resource manager who also plays this role. It is very useful for them to be accustomed to the practices, ideas, and values involved in managing a project. And they need to appreciate what the project manager contributes, so as to work cooperatively with them readily when they start their careers.

More generally, this course includes most of the topics about software process and about professional practice.

What is the format of the course?

The course is taught as an hour discussion / work activity 4 times a week. There is some lecture, but this is not a dominant part of even discussion times. The goal of every class session is for individual students and teams to be able to apply requirements-related and other skills as soon as possible, with growing independence, and to learn from that application.

How are students assessed?

Homework Assignments – Homework assignments performed throughout the term are used to apply and reinforce material from the course lectures and discussions. Homework assignments also encompass case studies discussed in class to illustrate the importance of project management in determining project success and failure.

Project - Each student will have a project and a team to work with on this. We have taught this class both coordinated with the full-year junior project, and with separate projects.

Exams – Two exams are used to test the students' knowledge and capabilities regarding the course material.

Quizzes - Daily quizzes completed during class.

Course Assessment Matrix

Assessment Tool:	**Outcome:**

	1	2	3	4	5	6	7	8
Project				X	X	X	X	X
Homework			X	X	X	X		
Exams	X	X	X				X	
Quizzes	X	X						X

Success Criteria

The course will be considered fully successful if the following statement holds for every tool-outcome pair selected above:

Among the students who earn proficient grades in the course, the average grade on the portions of the assessment tools that are relevant to the learning outcome is in the proficient range.

Course textbooks and materials

- *Agile Project Management,* Second Edition, by Jim Highsmith.
- *The Software Project Manager's Handbook*, SecondEdition, by Dwayne Phillips

Both textbooks are required, and discussions and assignments come from them directly.

Pedagogical advice

The course teaches both traditional engineering project management and agile processes, using the two textbooks. Clearly, it's important to keep separate these two alternatives, their terminology, and their strengths and weaknesses.

Body of Knowledge coverage

The course meets for 10 weeks, plus a final exam week. So there are 4 hours per week times 10, or 40 "contact hours" total. The 40 available hours are shown divided up in the table below.

KA	**Topic**	**Hours**
FND	**Mathematical and Engineering Fundamentals**	**8 Total**
FND.ec	Engineering economics for software	**8 total**
FND.ec.1	Value considerations throughout the software life cycle	4
FND.ec.2	Evaluating cost-effective solutions (e.g. benefits realization, tradeoff analysis, cost analysis, return on investment, etc.)	4
PRO	**Software Process**	**20 Total**

PRO.con	Process concepts	**2 total**
PRO.con.1	Themes and terminology	.5
PRO.con.2	Software engineering process infrastructure (e.g. personnel, tools, training, etc.)	.5
PRO.con.3	Software engineering process improvement (individual, team, organization)	.5
PRO.con.4	Systems engineering life cycle models	.5
PRO.imp	Process implementation	**5 total**
PRO.imp.1	Levels of process definition (e.g. organization, project, team, individual, etc.)	1
PRO.imp.2	Life cycle model characteristics (e.g., plan-based, incremental, iterative, agile)	1
PRO.imp.3	Individual and team software process (model, definition, measurement, analysis, improvement)	1
PRO.imp.4	Software process implementation in the context of systems engineering	1
PRO.imp.5	Effect of external factors (e.g., contract and legal requirements, standards, acquisition practices) on software process	1
PRO.pp	Project planning and tracking	**7 total**
PRO.pp.1	Requirements management (e.g., product backlog, priorities, dependencies, changes)	1
PRO.pp.2	Effort estimation (e.g., use of historical data, consensus-based estimation techniques)	1
PRO.pp.3	Work breakdown and task scheduling	1
PRO.pp.4	Resource allocation	1
PRO.pp.5	Risk management (e.g., identification, mitigation, remediation, status tracking)	1
PRO.pp.6	Project tracking metrics and techniques (e.g., earned value, velocity, burndown charts, defect tracking, management of technical debt)	1
PRO.pp.7	Team self-management (e.g., progress tracking, dynamic workload allocation, response to emergent issues)	1
PRO.cm	Software configuration management	**6 total**
PRO.cm.1	Revision control	1

PRO.cm.2	Release management	1
PRO.cm.3	Configuration management tools	1
PRO.cm.4	Build processes and tools, including automated testing and continuous integration	1
PRO.cm.5	Software configuration management processes	1
PRO.cm.6	Software deployment processes	.5
PRO.cm.7	Distribution and backup	.5
PRF	**Professional Practice**	**12 Total**
PRF.psy	Group dynamics / psychology	**4 total**
PRF.psy.1	Dynamics of working in teams/groups	1
PRF.psy.2	Interacting with stakeholders	1
PRF.psy.3	Dealing with uncertainty and ambiguity	1
PRF.psy.4	Dealing with multicultural environments	1
PRF.com	Communications skills (specific to SE)	**4 total**
PRF.com.1	Reading, understanding and summarizing reading (e.g. source code, documentation)	1
PRF.com.2	Writing (assignments, reports, evaluations, justifications, etc.)	1
PRF.com.3	Team and group communication (both oral and written)	1
PRF.com.4	Presentation skills	1
PRF.pr	Professionalism	**4 total**
PRF.pr.1	Accreditation, certification, and licensing	.5
PRF.pr.2	Codes of ethics and professional conduct	.5
PRF.pr.3	Social, legal, historical, and professional issues and concerns	.5
PRF.pr.4	The nature and role of professional societies	.5
PRF.pr.5	The nature and role of software engineering standards	.5
PRF.pr.6	The economic impact of software	.5
PRF.pr.7	Employment contracts	.5
PRF.pr.8	Intellectual property, software licensing and contracts	.5

Additional topics

Students are expected to participate in course improvement. This means getting their feedback, and taking pre and post-course questionnaires regarding their level of understanding of course topics, among other things.

Other comments

(none)

B.4. Formal Methods (RHIT)

CSSE 373, Formal Methods
Rose-Hulman Institute of Technology, Terre Haute, IN, USA
Instructors: Chandan Rupakheti
Email Addresses: rupakhet@rose-hulman.edu
URL for additional information:

Catalog description
Introduction to the use of mathematical models of software systems for their specification and validation. Topics include finite state machine models, models of concurrent systems, verification of models, and limitations of these techniques.

Expected Outcomes

Students who complete this course will be able to:

1 demonstrate formal correctness of simple procedure

2 construct formal models of sequential software systems

3 implement sequential software systems based on formal models

4 verify attributes of formal models

5 describe the costs and benefits of formal methods.

Where the course fits into our curriculum

Normally taught in:
Spring of junior year for most students, though it can be delayed to senior year.

Course Prerequisites:
CSSE 230 (Fundamentals of Software Development III, our data structures course) or equivalent, and MA275 Discrete and Combinatorial Algebra I).

Normally followed immediately by:
CSSE 497 (Senior Project - 1), the following fall. Also, by a software-related internship in the summer in-between.

What is covered in the course?

The course is primarily about rigorous methods of requirements, design, and code analysis.

Many students do not use these in their work after graduation, which also is true of other required courses like calculus and physics, not to mention the content of more analytical computer science courses like data structures. Some do, however, and for the rest we believe that the course gives students a deeper understanding about what it takes to know the correctness of the processes they will use.

What is the format of the course?

The course is taught as an hour discussion / work activity 4 times a week. There is some lecture, but this is not a dominant part of even discussion times. The goal of every class session is for individual students and teams to be able to apply requirements-related and other skills as soon as possible, with growing independence, and to learn from that application.

How are students assessed?

Homework Assignments - exercises in the application of formal methods

Project - Team project in specification and analysis of a safety-critical application, with short individual essay at completion.

Exams - three one-hour exams.

COURSE ASSESSMENT MATRIX

	Outcome				
	1	2	3	4	5
Homework	X	X	X	X	X
Project		X	X		X
Exams	X			X	

SUCCESS CRITERIA

The course will be considered fully successful if the following statement holds for every tool-outcome pair selected above:

Among the students who earn proficient grades in the course, the average grade on the portions of the assessment tools that are relevant to the learning outcome is in the proficient range.

Course textbooks and materials

Please contact the instructor, *above*, for current information about resources used.

Pedagogical advice

Body of Knowledge coverage

The course meets for 10 weeks, plus a final exam week. So there are 4 hours per week times 10, or 40 "contact hours" total. The 40 available hours are shown divided up in the table below.

KA	Topic	Hours
MAA	**Software Modeling and Analysis**	**16 Total**
MAA.af	Analysis fundamentals	**6 total**
MAA.af.1	Analyzing well-formedness (e.g. completeness, consistency, robustness, etc.)	1
MAA.af.2	Analyzing correctness (e.g. static analysis, simulation, model checking, etc.)	1
MAA.af.3	Formal analysis	4
MAA.md	Modeling foundations	**10 total**
MAA.md.1	Modeling principles (e.g. decomposition, abstraction, generalization, etc.)	3
MAA.md.2	Preconditions, postconditions, invariants, contracts	4
MAA.md.3	Introduction to mathematical models and formal notation	3
MAA.tm	Types of models	**0 total**
MAA.tm.1	Information modeling (e.g. entity-relationship modeling, class diagrams, etc.)	0 (covered in CSSE 333)
MAA.tm.2	Behavioral modeling (e.g. use case analysis, activity diagrams, interaction diagrams, state machine diagrams, etc.)	0 (covered in CSSE 371)
MAA.tm.3	Structure modeling (e.g. class diagrams, etc.)	0 (covered in CSSE 374)
MAA.tm.4	Domain modeling (e.g. domain engineering approaches, etc.)	0 (covered in CSSE 371)
MAA.tm.5	Functional modeling (e.g. component diagrams, etc.)	0 (covered in CSSE 374)
MAA.tm.6	Enterprise modeling (e.g. business processes,	0 (not

	organizations, goals, etc.)	covered)
MAA.tm.7	Modeling embedded systems (e.g. real-time schedulability analysis, external interface analysis, etc.)	0 (covered in Operating Systems)
DES	**Software Design**	**11 Total**
DES.dd	Detailed design	**5 total**
DES.dd.4	Design notations (e.g. class and object diagrams, UML, state diagrams, formal specification, etc.)	5
DES.ev	Design evaluation	**6 total**
DES.ev.3	Formal design analysis	6
VAV	**Software verification and validation**	**3 Total**
VAV.rev	Reviews and static analysis	**3 total**
VAV.rev.3	Static analysis (common defect detection, checking against formal specifications, etc.)	3
REQ	**Requirements analysis and specification**	**5 Total**
REQ.rv	Requirements validation	**5 total**
REQ.rv.6	Formal requirements analysis	5
SEC	**Security**	**5 Total** (interpreted here also to include Safety)
SEC.dev	Developing secure software	**5 total**
SEC.dev.1	Building security into the software development life cycle	1
SEC.dev.4	Secure software construction techniques	1
SEC.dev.5	Security-related verification and validation	3

Additional topics

Students are expected to participate in course improvement. This means getting their feedback, and taking pre and post-course questionnaires regarding their level of understanding of course topics, among other things.

Other comments

(none)

B.5. Software Design (RHIT)

CSSE 374, Software Design
Rose-Hulman Institute of Technology, Terre Haute, IN, USA
Instructors: Steve Chenoweth, Chandan Rupakheti, Shawn Bohner
Email Addresses: chenowet@rose-hulman.edu, rupakhet@rose-hulman.edu, bohner@rose-hulman.edu
URL for additional information: http://www.rose-hulman.edu/class/csse/csse374/

Catalog description
Introduction to the design of complete software systems, building on components and patterns. Topics include architectural principles and alternatives, design documentation, and relationships between levels of abstraction.
Expected Outcomes

Students that successfully complete the course will be able to:

1. Assess and improve the effectiveness of a team of software project stakeholders, including customers, users, and members of a significantly sized development overall team that is made up of smaller teams and using cross-teams.
2. Recognize the differences between problems and solutions, and deal with their interactions.
3. Use agile methods to design and develop a system for a real customer.
4. Demonstrate object-oriented design basics like domain models, class diagrams, and interaction (sequence and communication) diagrams.
5. Use fundamental design principles, methods, patterns and strategies in the creation of a software system and its supporting documents.
6. Identify criteria for the design of a software system and select patterns, create frameworks, and partition software to satisfy the inherent trade-offs.
7. Analyze and explain the feasibility and soundness of a software design.

Where the course fits into our curriculum

Normally taught in:
Winter of junior year for almost all students.

Course Prerequisites:
CSSE 371 (Software Requirements Engineering), which has as its prerequisites CSSE 230 (Fundamentals of Software Development III, our data structures course) or equivalent; RH 330 or equivalent (our second technical writing course); and Junior standing. The latter is important, because many students are able to get a summer internship after their sophomore year. That is often effective in increasing their interest in software engineering practices and, sometimes, in the value of getting a good design the first time, and use of OO design methods.

Normally this course follows:

CSSE 371 and RH 330 (see above), and CSSE 376 (software quality assurance).

Normally followed immediately by:
CSSE 375 (Software Construction and Evolution) for software engineering majors.

Also required for:
CSSE 498-9 (final two terms of the 3-term Senior Project sequence)

What is covered in the course?

This is a course in OO design, with emphasis especially on the Gang of Four software patterns and creation of frameworks that enable development using SOLID principles. A heavy component of the course is developing a large project for which use of the best patterns, single responsibility principle, dependency inversion, etc. enables easier extensions later on.

The course also is the second of three during the junior year, for software engineering majors, which involve developing the project. That project is completed by software engineering majors in CSSE 375, during the spring term.

The intent is that the main learning is all experiential, from students' trying design ideas in different situations, to learn first-hand what works and what doesn't. It builds on the Rose-Hulman theme that, as much as possible, students should learn by doing, and not just by studying about topics. It also is anticipated that they will experience failure part of the time and, indeed, their project may not be progressing successfully at any given point, at the end of one of the three courses, or even overall. It is, after all, their first larger-than-class-sized project.

What is the format of the course?

The course is taught as a two hour discussion / work activity / lab 3 times a week, for a total of 6 hours per week. There is some lecture, but this is not a dominant part of even discussion times. The goal of every class session is for individual students and teams to be able to apply design-related and other skills as soon as possible, with growing independence, and to learn from that application. There are in-class exercises, homeworks, and the project toward this end.

How are students assessed?

Homework - Weekly exercises on course material and elements of project.

Project – The middle 10 weeks of a 30 week software team project, producing a software design document and an executable framework using patterns, as well as related deliverables for a customer external to the team. The project used to be for individual teams of 3 or 4 students. It is now a full section-sized (20 student) project, sub-divided

into smaller teams with specific roles (like different feature sets or implementation on different devices).

Journals – As an integral part of the project, students are expected to keep technical journals in which they record both team and individual activities that were a part of their work. These are intended, in particular, to demonstrate that the students did critical thinking as a part of the project problem solving. Thus, the journals are graded on this basis, and cannot simply log what happened without showing a search for root causes, development of creative ideas, reflection on teaming experiences, and how they made personal contributions to the team. Along with other ways to judge individual contributions on teams, these journals can be used as a subjective means to verify the significance of those contributions.

Project team meetings – Regular meetings with the instructor to review progress on software projects (typically multiple times a week), and client meetings (t (at least every other week) which the instructor observes.

Exams/Quizzes - two exams.

Quizzes – daily (done before or during class, on the material for that session).

Course Assessment Matrix

	Outcome					
	1	2	3	4	5	6
Homework		X		X		
Project	X		X			X
Project team meetings	X		X			X
Exams		X			X	
Quizzes		X		X	X	

Success Criteria

The course will be considered fully successful if the following statement holds for every tool-objective pair selected above:

Among the students who earn proficient grades in the course, the average grade on the portions of the assessment tools that are relevant to the learning objective is in the proficient range.

Course textbooks and materials

Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3ed) by Craig Larman; Prentice Hall PTR (2004). ISBN 13: 978-0131489066.

Readings may also be assigned from other relevant technical publications. E.g., readings on coupling and cohesion; and readings on the "SOLID" principles, such as http://lostechies.com/wp-content/uploads/2011/03/pablos_solid_ebook.pdf.

Pedagogical advice

For an OO design course with a large, integral project, it is strategic to have guessed correctly, that the project will in fact benefit from many of the GoF patterns and SOLID principles. Since we do a new and different project every school year, this isn't easy to get right. The fallback position is that patterns and frameworks could be employed in the project, so as to make it more flexible and extendable, if the client didn't necessarily ask for such a general project in her requirements.

Clients must be found who are willing to dedicate a larger amount of their time than usually is expected even for a senior design team. As an example of the sorts of planning that need to be done, clients normally should be available to meet with the sections representing their project, at one of the times that section is scheduled!

We used SCRUM, with two-week sprints. Thus, having course material in the three courses progress at the usual pace did not fit this agile method. For this design course, that means that testing and delivery of successive sprints needs to be covered, or taken previously.

Body of Knowledge coverage

Note that the "contact hours" listed in the right-hand column are a rather rubbery number. We all see this in senior design courses, because it is self-regulated and projects differ in the amount of work of each type. In this design course, the major project is a similar source of variation. While the course provides more guidance than is true in senior design, the goal is for students to do as much as possible, on teams, on their own. The course meets for 10 weeks, plus a final exam week. So there are 6 hours per week times 10, or 60 "contact hours" total. The 60 available hours are shown divided up in the table below.

KA	Topic	Hours
DES	**Software Design**	26 Total
DES.con	Design concepts	3 total
DES.con.1	Definition of design	.5
DES.con.2	Fundamental design issues (e.g. persistent data, storage management, exceptions, etc.)	.5
DES.con.3	Context of design within multiple software development life cycles	.5

DES.con.4	Design principles (information hiding, cohesion and coupling)	.5
DES.con.5	Design for quality attributes (e.g. reliability, usability, maintainability, performance, testability, security, fault tolerance, etc.)	.5
DES.con.6	Design trade-offs	.5
DES.str	Design strategies	6 total
DES.str.1	Function-oriented design	0 (done in prior courses)
DES.str.2	Object-oriented design	6
DES.str.3	Data-structure centered design	0 (done in prerequisite CSSE 230)
DES.str.4	Aspect-oriented design	0 (not included)
DES.ar	Architectural design	0 (done in following course, CSSE 477, or in preceding course CSSE 371)
DES.dd	Detailed design	14 total
DES.dd.1	Design patterns	10
DES.dd.2	Database design	0 (done in required database course CSSE 333)
DES.dd.3	Design of networked and mobile systems	0 (not done unless required for project)
DES.dd.4	Design notations (e.g. class and object diagrams, UML, state diagrams, formal specification, etc.)	4
DES.ev	Design evaluation	3 total
DES.ev.1	Measures of design attributes (e.g. coupling, cohesion, information-hiding, separation of concerns, etc.)	3
PRF	**Professional Practice**	**23 Total**

PRF.psy	Group dynamics / psychology	11 total
PRF.psy.1	Dynamics of working in teams/groups	4
PRF.psy.2	Interacting with stakeholders	4
PRF.psy.3	Dealing with uncertainty and ambiguity	3
PRF.psy.4	Dealing with multicultural environments	0 (depends on the project)
PRF.com	Communications skills (specific to SE)	12 total
PRF.com.1	Reading, understanding and summarizing reading (e.g. source code, documentation)	2
PRF.com.2	Writing (assignments, reports, evaluations, justifications, etc.)	4
PRF.com.3	Team and group communication (both oral and written)	4
PRF.com.4	Presentation skills	2
MAA	**Software Modeling and Analysis**	**6 Total**
MAA.tm	Types of models	1
MAA.tm.3	Structure modeling (e.g. class diagrams, etc.)	5
VAV	**Software verification and validation**	**5 Total**
VAV.rev	Reviews and static analysis	1
VAV.rev.1	Personal reviews (design, code, etc.)	1
VAV.rev.2	Peer reviews (inspections, walkthroughs, etc.)	3

Additional topics

Students are expected to participate in course improvement. This means getting their feedback, and taking pre and post-course questionnaires regarding their level of understanding of course topics, among other things.

Other comments

Note the fact that this course continues with the middle part of a 3-course project. This means, among other things, that students will end the course not having the clean feeling of completing something delivered to the instructor or to a customer. This tends to alter the types of comments we get from them at the end of the course. Many software students

rate the value of their own learning on having been successful in finishing the major class project, and they won't get that finality here.

In trade for that, students get to work on projects which are large enough that the software practices do make a difference in success of the project, something they will lose if the project is small and all the non-coding parts are felt to be trivial.

B.6. Software Construction & Evolution (RHIT)

CSSE 375, Software Construction & Evolution
Rose-Hulman Institute of Technology, Terre Haute, IN, USA
Instructors: Michael Hewner, Shawn Bohner, Steve Chenoweth
Email Addresses: hewner@rose-hulman.edu, bohner@rose-hulman.edu, chenowet@rose-hulman.edu
URL for additional information: http://www.rose-hulman.edu/class/csse/csse375/

Catalog description
Issues, methods and techniques associated with constructing software. Topics include detailed design methods and notations, implementation tools, coding standards and styles, peer review techniques, and maintenance issues.

Expected Outcomes

Students that successfully complete the course should be able to:

1. Work with junior project team to complete and deliver the junior project to the client. In doing so, demonstrate the ability to work within a team to deliver a multi-term-sized project to an external client successfully.
2. Apply appropriate refactoring techniques to resolve design problems in code.
3. Apply common construction and maintenance heuristics to enhance existing code, such as ways to eliminate global variables and ways to test difficult code.
4. Organize and develop software user documentation which enhances long-term software viability.
5. Construct software so that it meets delivery and deployment objectives specified by the project.
6. Apply the corrective, perfective, adaptive and preventive types of software changes and maintenance types.
7. Apply impact analysis and other software source analysis to understanding existing software.
8. Use systematic exception handling and other techniques in promoting fault-tolerance.
9. Describe software modernization approaches such as reverse engineering, reengineering, salvaging, and restructuring.
10. Describe the ways configuration management is used in production systems.

Where the course fits into our curriculum

Normally taught in:
Spring of junior year for almost all students.

Course Prerequisites:

CSSE 374 (Software Design), which has as its prerequisite CSSE 371 (Software Requirements Engineering), which has as its prerequisites CSSE 230 (Fundamentals of Software Development III, our data structures course) or equivalent; RH 330 or equivalent (our second technical writing course); and Junior standing.

Normally this course follows:
CSSE 374, and prior work on the same junior project.

Normally followed by:
CSSE 497-8-9, Senior Project.

What is covered in the course?

One of the places where most computer science programs miss the mark completely is in having students do all "greenfield systems," all the time. By the time they are seniors, they seriously believe the solution to anything is to rewrite it completely, themselves.

In industry this inclination will get you fired. Developers will build on top of other software, or maintain existing software, all the time. Thus, understanding and revising other people's designs and coding are strategic skills. This course is about those topics.

The course begins with the application of Martin Fowler's refactoring ideas to multiple projects, in homework programs and in the junior project. Regarding the latter, students have been working on it for two full terms already, and began coding back in the first term, without guidance about refactoring as a part of the development, so there is plenty there to refactor by now!

The second major section involves applying Michael Feathers' "legacy code" concepts. Once again, the students' own ongoing large project is a perfect target for these. There is probably plenty of code there which is hard to unit test and hard to enhance.

We added a significant section on exception handling, which is an area that students notoriously are under-educated about when they enter industry. The topics include, for example, making methods robust by having them check their inputs sent from calling objects.

The course also includes standard topics about construction and maintenance, such as Lehman's Laws, code salvaging, and configuration management. Students by now have had enough development experience that they can relate to most of these subjects.

What is the format of the course?

The course is taught as a one hour discussion / work activity class three times a week, plus a 3-hour lab once a week. There is some lecture, but this is not a dominant part of even discussion times. The goal of every class session is for individual students and teams to be able to apply construction skills as soon as possible, with growing

independence, and to learn from that application. There are in-class exercises, homeworks, and the project toward this end.

How are students assessed?

Homework – Homework assignments performed throughout the term are used to apply and reinforce material from the course lectures and discussions. This includes the use of an approach where programming assignments are swapped between students to review and add features.

Project Deliverables - Each student is part of a team project with an external entity or Rose-Hulman faculty or staff member as a client; each team has a different project (also common to CSSE 371 and 374). The project applies the methods and technology to the junior project sequence.

Project Participation - Each student is part of a team project where they are integral to the success of the team. Based on student peer evaluations and instructor observations, the student's contribution to the overall project is assessed.

Journals – As an integral part of the project, students are expected to keep technical journals in which they record both team and individual activities that were a part of their work. These are intended, in particular, to demonstrate that the students did critical thinking as a part of the project problem solving. Thus, the journals are graded on this basis, and cannot simply log what happened without showing a search for root causes, development of creative ideas, reflection on teaming experiences, and how they made personal contributions to the team. Along with other ways to judge individual contributions on teams, these journals can be used as a subjective means to verify the significance of those contributions.

Exams – Two exams (one mandatory and one optional) are used to test the students' knowledge and capabilities regarding the course material.
Quizzes - Daily quizzes completed during class to cover learning objectives.

Course Assessment Matrix
Assessment will be done differently than last years due to the fact we will have different assignments and rubrics. Since we do not have TAs, we will be using Senior Project teams to do the Project Delivery Review and advisement.

	Learning Outcome:									
Assessment Tool:	1	2	3	4	5	6	7	8	9	10
Homework		X	X							
Project Deliverable	X	X		X	X					X
Project Participation	X			X	X					X

Exams						X	X	X	X	
Quizzes		X	X			X	X	X	X	

Success Criteria

The course will be considered fully successful if the following statement holds for every tool-outcome pair selected above:

Among the students who earn proficient grades in the course, the average grade on the portions of the assessment tools that are relevant to the learning outcome is in the proficient range.

Course textbooks and materials

- *Working Effectively with Legacy Code*, by Michael C. Feathers. Publisher: Pearson Education, Prentice-Hall ISBN-10: 0-13-117705-2

- *Refactoring: Improving the Design of Existing Code*, by Martin Fowler Publisher: Addison-Wesley Professional; 1 edition (July 8, 1999) ISBN-10: 0201485672

Pedagogical advice

This course includes delivery of a system that students will have worked on for three terms. The intent of the course is to teach the topics described, yet it is done via problem-based learning, so there could be variances between the course expectations and the client's expectations. For example, the client may not care if a completed "spec" accompanies the code they receive. While students may perceive this conundrum as artificial, it does have an analogy in industry: Software development shops each have their own "standards," and those may or may not coincide with their clients' standards, for example, when one is delivering to another software organization.

Body of Knowledge coverage

Note that the "contact hours" listed in the right-hand column are a rather rubbery number. We all see this in senior design courses, because it is self-regulated and projects differ in the amount of work of each type. In this construction course, the major project is a similar source of variation. While the course provides more guidance than is true in senior design, the goal is for students to do as much as possible, on teams, on their own. The course meets for 10 weeks, plus a final exam week. So there are 6 hours per week times 10, or 60 "contact hours" total. The 60 available hours are shown divided up in the table below.

KA	Topic	Hours
PRO	**Software Process**	**46 Total**

PRO.con	Process concepts	6 total
PRO.con.1	Themes and terminology	1
PRO.con.2	Software engineering process infrastructure (e.g. personnel, tools, training, etc.)	1
PRO.con.3	Software engineering process improvement (individual, team, organization)	2
PRO.con.4	Systems engineering life cycle models	2
PRO.imp	Process implementation	4 total
PRO.imp.1	Levels of process definition (e.g. organization, project, team, individual, etc.)	1
PRO.imp.2	Life cycle model characteristics (e.g., plan-based, incremental, iterative, agile)	1
PRO.imp.3	Individual and team software process (model, definition, measurement, analysis, improvement)	1
PRO.imp.4	Software process implementation in the context of systems engineering	0 (unless the project has significant hardware concerns)
PRO.imp.5	Effect of external factors (e.g., contract and legal requirements, standards, acquisition practices) on software process	1
PRO.pp	Project planning and tracking	0 (covered in the Requirements Engineering and Project Management courses)
PRO.cm	Software configuration management	4 total (many parts covered in prior courses)
PRO.cm.2	Release management	1
PRO.cm.5	Software configuration management processes	1
PRO.cm.6	Software deployment processes	1
PRO.cm.7	Distribution and backup	1
PRO.evo	Evolution processes and activities	32 total

PRO.evo.1	Basic concepts of evolution and maintenance	4
PRO.evo.2	Working with legacy systems	12
PRO.evo.3	Refactoring	16
CMP	**Computing essentials**	**10 Total**
CMP.cf	Computer science foundations	4 total
CMP.cf.6	Basic user human factors (I/O, error messages, robustness)	2
CMP.cf.7	Basic developer human factors (comments, structure, readability)	2
CMP.ct	Construction technologies	5 total
CMP.ct.1	API design and use	.5
CMP.ct.2	Code reuse and libraries	.5
CMP.ct.6	Error handling, exception handling, and fault tolerance	2
CMP.ct.7	State-based and table-driven construction techniques	0 (unless the project requires this)
CMP.ct.8	Run-time configuration and internationalization	.5
CMP.ct.11	Construction methods for distributed software (e.g., cloud and mobile computing)	.5
CMP.ct.13	Debugging and fault isolation techniques	1
CMP.tl	Construction tools	1 total
CMP.tl.2	User interface frameworks and tools	1
VAV	**Software verification and validation**	**4 Total**
VAV.rev.1	Personal reviews (design, code, etc.)	.5
VAV.rev.2	Peer reviews (inspections, walkthroughs, etc.)	3.5

Additional topics

Students are expected to participate in course improvement. This means getting their feedback, and taking pre and post-course questionnaires regarding their level of understanding of course topics, among other things.

Other comments

Note the fact that this course completes a 3-course project. This means, among other things, that the success or failure of the project, of which this course was only a part, will weigh heavily on students, as they decide what they learned (because so many consider project success to mean they learned the material!).

In trade for that, students get to work on projects which are large enough that the software practices do make a difference in success of the project, something they will lose if the project is small and all the non-coding parts are felt to be trivial.

B.7. Software Quality Assurance (RHIT)

CSSE 376, Software Quality Assurance
Rose-Hulman Institute of Technology, Terre Haute, IN, USA
Instructors: Michael Hewner, Sriram Mohan
Email Addresses: hewner@rose-hulman.edu, mohan@rose-hulman.edu
URL for additional information:

Catalog description
Theory and practice of determining whether a product conforms to its specification and intended use. Topics include software quality assurance methods, test plans and strategies, unit level and system level testing, software reliability, peer review methods, and configuration control responsibilities in quality assurance.

Expected Outcomes

Students who complete this course will be able to:

1. Create a test plan for a software system

2. Apply different strategies for unit-level and system-level testing

3. Apply principles and strategies of integration and regression testing

4. Explain purposes of metrics, quality processes, methods for measuring that quality, and standards used

5. Apply principles of test driven development to successfully develop a software product

Where the course fits into our curriculum

Normally taught in:
Spring of sophomore year for almost all students.

Course Prerequisites:
CSSE 230 (Fundamentals of Software Development III, our data structures course) or equivalent.

Normally this course is taken at the same time as:
RH330 (see above), and a course in the software engineering major's domain track.

Normally followed immediately by:
CSSE 371 (Software Requirements Engineering), the following fall. Also, by a software-related internship in the summer in-between.

What is covered in the course?

The course is primarily about testing, versus creating quality by processes preceding testing.

Many of our students start their careers, after graduation, with a job in QA. This course is specific training for that position.

What is the format of the course?

The course is taught as an hour discussion / work activity 4 times a week. There is some lecture, but this is not a dominant part of even discussion times. The goal of every class session is for individual students and teams to be able to apply requirements-related and other skills as soon as possible, with growing independence, and to learn from that application.

How are students assessed?

Labs - A series of labs in which the students learn to plan and conduct testing of software.

Project - A team of students will use the principles of Test Driven Development on a five week software development exercise

DT Presentation - A team of students will choose from one of the several domain tracks that are offered by Rose-Hulman and describe Quality assurance practices that are in vogue.

Exams - two one-hour exams.

Course Assessment Matrix

	Objective				
	1	2	3	4	5
Labs	X	X	X	X	X
Project	X	X	X	X	X
Exams					
DT Presentation				X	

Success Criteria

The course will be considered fully successful if the following statement holds for every tool-objective pair selected above:

Among the students who earn proficient grades in the course, the average grade on the portions of the assessment tools that are relevant to the learning objective is in the proficient range.

Course textbooks and materials
The course is taught without a textbook, largely as a series of labs in these areas:

1. Software Craftsmanship
2. GIT Basics
3. Unit Testing
4. Test Driven Development
5. Code Coverage
6. Mocking
7. Integration Testing
8. Performance Testing
9. Localization
10. Metrics
11. Test Plans
12. Behavior Driven Development

Pedagogical advice

Body of Knowledge coverage

The course meets for 10 weeks, plus a final exam week. So there are 4 hours per week times 10, or 40 "contact hours" total. The 40 available hours are shown divided up in the table below.

KA	Topic	Hours
VAV	**Software verification and validation**	**20 Total**
VAV.fnd	V&V terminology and foundations	**3 total**
VAV.fnd.1	Objectives and constraints of V&V	1
VAV.fnd.2	Metrics & measurement (e.g. reliability, usability, performance, etc.)	1
VAV.fnd.3	V&V involvement at different points in the life cycle	1
VAV.rev	Reviews and static analysis	0 (included in CSSE 375)
VAV.tst	Testing	**14 total**
VAV.tst.1	Unit testing and test-driven development	1
VAV.tst.2	Stress testing	1
VAV.tst.3	Criteria-based test design (e.g., graph-based, control flow coverage, logic coverage)	1

VAV.tst.4	Model-based test design (e.g., UML diagrams, state charts, sequence diagrams, use cases)	1
VAV.tst.5	Human-based testing (e.g., black box testing, domain knowledge)	1
VAV.tst.6	Integration testing	1
VAV.tst.7	System testing	1
VAV.tst.8	Acceptance testing	1
VAV.tst.9	Testing across quality attributes (e.g. usability, security, compatibility, accessibility, performance etc.)	1
VAV.tst.10	Regression testing	1
VAV.tst.11	Testing tools	3
VAV.tst.12	Test automation (e.g., test scripts, interface capture/replay, unit testing)	1
VAV.par	Problem analysis and reporting	**3 total**
VAV.par.1	Analyzing failure reports	1
VAV.par.2	Root-cause analysis (e.g., identifying process or product weaknesses that promoted injection or hindered removal of serious defects)	1
VAV.par.3	Problem tracking	1
QUA	**Software Quality**	**20 Total**
QUA.cc	Software quality concepts and culture	**9 total**
QUA.cc.1	Definitions of quality	1
QUA.cc.2	Society's concern for quality	1
QUA.cc.3	The costs and impacts of bad quality	1
QUA.cc.4	A cost of quality model	2
QUA.cc.5	Quality attributes for software (e.g. dependability, usability, safety, etc.)	2
QUA.cc.6	Roles of people, processes, methods, tools, and technology	2
QUA.pca	Process assurance	**5 total**
QUA.pca.1	The nature of process assurance	1
QUA.pca.2	Quality planning	1

QUA.pca.3	Process assurance techniques	3
QUA.pda	Product assurance	**6 total**
QUA.pda.1	The nature of product assurance	1
QUA.pda.2	Distinctions between assurance and V&V	1
QUA.pda.3	Quality product models	1
QUA.pda.4	Root cause analysis and defect prevention	1
QUA.pda.5	Quality product metrics and measurement	1
QUA.pda.6	Assessment of product quality attributes (e.g. usability, reliability, availability, etc.)	1

Additional topics

Students are expected to participate in course improvement. This means getting their feedback, and taking pre and post-course questionnaires regarding their level of understanding of course topics, among other things.

Other comments

(none)

B.8. Software Architecture (RHIT)

CSSE 477, Software Architecture
Rose-Hulman Institute of Technology, Terre Haute, IN, USA
Instructors: Chandan Rupakheti, Steve Chenoweth
Email Addresses: rupakhet@rose-hulman.edu, chenowet@rose-hulman.edu
URL for additional information: http://www.rose-hulman.edu/class/csse/csse477/
(Note: This version is circa 2011-12.)

Catalog description
This is a second course in the architecture and design of complete software systems, building on components and patterns. Topics include architectural principles and alternatives, design documentation, relationships between levels of abstraction, theory and practice of human interface design, creating systems which can evolve, choosing software sources and strategies, prototyping and documenting designs, and employing patterns for reuse. How to design systems which a team of developers can implement, and which will be successful in the real world.

Expected Outcomes

Students who complete this course successfully should be able to:

1. Design and build effective human-computer interfaces using standard methods and criteria. (Extending what's in CSSE 371 on this subject).

2. Describe the basic ingredients of successful software product lines – How to do multiple releases of software.

3. Analyze the quality attributes, economics and other global properties of existing designs and systems, and gain experience building systems so as to have desirable global properties. This is the heart of software architecture. Includes also make vs. buy decisions, and discussion of component selection.

4. Create and document the overall design for a system and document this design using UML and other methodologies and notations. This elaborates on the ways to develop, prototype and document architectures.

5. Practice the process by which architectures get created, in terms of technologies, economics, people, and processes – Extends the project work of CSSE 374, looking at more patterns and new angles, including also some full-blown design methods like use of different architectural styles.

6. Describe the basic structure and functioning of systems using Service Oriented Architecture (SOA).

Where the course fits into our curriculum

Normally taught in:
Fall of senior year for almost all students.

Course Prerequisites:
CSSE 374 (Software Design), which has as its prerequisite CSSE 371 (Software Requirements Engineering), which has as its prerequisites CSSE 230 (Fundamentals of Software Development III, our data structures course) or equivalent; RH 330 or equivalent (our second technical writing course); and Junior standing.

Normally this course follows:
The entire junior sequence, CSSE 371, 374 and 375, plus probably 373 and, in the sophomore year, 376.

Normally coincides with:
CSSE 497, the first course of three in Senior Project. Thus, students are learning about architectural styles, etc., immediately before applying that knowledge to their own senior project.

What is covered in the course?

A major lesson of the course is for students to learn how to provide architectural attributes in their designs, alias quality attributes, alias non-functional attributes. Bass, et al's book is used because it teaches scenarios for this, how to put in place a way of expressing what is needed, which can grow into what is designed and implemented and tested, as is true for functional attributes with use cases.

What is the format of the course?

The course is taught as a one hour discussion / work activity class four times a week. There is some lecture, but this is not a dominant part of even discussion times. The goal of every class session is for individual students and teams to be able to apply construction skills as soon as possible, with growing independence, and to learn from that application. There are in-class exercises, homeworks, and the project toward this end.

How are students assessed?

Team Projects – This is the major student deliverable in the course, a full-term project done by teams of two peers.

The project is chosen by each team, often as a continuation of work started in CSSE 371 and continued throughout the junior year in 374 and 375. In the 377 project the students will go through six architectural studies of that system, trying to improve on its quality attributes. These studies provide direct feedback on the difficulties in making such improvements after a system is already even partially built. The students also create an

accompanying software architecture document from scratch, after the fact, which tests their ability to capture a design (their own!) from its code. They continue with various design activities including application of patterns and frameworks, and make-versus-buy decisions.

The six major studies are done in a fashion that brings out the heuristic nature of architectural choices. For example, they test their ability to make changes to their system in a study of its modifiability. Then they try to refactor the system so as to improve that attribute systematically. Finally, they make a different set of changes to see if they actually improved the efficiency of maintenance.

Students will present and demonstrate these projects at the end of each of the six exercises. Their final presentation is an overall evaluation of what worked and what didn't.

Note: We have been experimenting with other kinds of projects the past couple years, including a larger, class-section-sized project for a single client.

Journals – As an integral part of the project, students are expected to keep technical journals in which they record both team and individual activities that were a part of their work. These are intended, in particular, to demonstrate that the students did critical thinking as a part of the project problem solving. Thus, the journals are graded on this basis, and cannot simply log what happened without showing a search for root causes, development of creative ideas, reflection on teaming experiences, and how they made personal contributions to the team. Along with other ways to judge individual contributions on teams, these journals can be used as a subjective means to verify the significance of those contributions.

Homework – These assignments are primarily individual ones such as answering questions at the ends of the chapters and small/mid-size projects to test concept comprehension. The exception is the presentation of an architecture case study, counted as homework, which requires a team of two students to do a full-hour presentation on one of the case histories from Bass's Software Architecture book. This assignment is clearly an application of earlier learning.

Biweekly quizzes – These are four short essay quizzes of approximately 30 minutes duration. All are closed book, done in class. The quizzes test for broad knowledge and application of concepts. A sample question is, "Ch 8 of Bass says that 'the complexity of flight simulators grew exponentially' over a 30 year period. Why was that particularly the case for this application? How would you allow for an application that doubled in size periodically?"

Term paper – The paper allows students to find a small, applied research topic in software architecture and analyze it as a team (of 2). A sample topic is, "Describe where the boundaries are on client-server designs, and what alternative architectural styles take

over at those boundary points." Student teams are allowed to come up with their own topics.

Course Assessment Matrix

	Objective					
	1	2	3	4	5	6
Team Project	X	X	X	X	X	
Homeworks	X	X	X	X		X
Biweekly Quizzes			X			X
Term Paper			X			

Success Criteria

The course will be considered fully successful if the following statement holds for every tool-outcome pair selected above:

Among the students who earn proficient grades in the course, the average grade on the portions of the assessment tools that are relevant to the learning outcome is in the proficient range.

Course textbooks and materials
Software Architecture in Practice, 3/E, by Len Bass, Paul Clements, and Rick Kazman

Pedagogical advice

Software architecture may be among the most difficult subjects to teach to students lacking in large system experience. Almost every project they have ever created runs fast enough and reliably enough on their own laptops, for instance. The idea is foreign to them, that making large systems work acceptably is a challenge, which may require rewriting the system if initial design choices are incorrect.

Body of Knowledge coverage

Note that the "contact hours" listed in the right-hand column are a rather rubbery number. We all see this in senior design courses, because it is self-regulated and projects differ in the amount of work of each type. In this design course, the major project, whatever it is, is a similar source of variation. While the course provides more guidance than is true in senior design, the goal is for students to do as much as possible, on teams, on their own. The course meets for 10 weeks, plus a final exam week. So there are 4 hours per week times 10, or 40 "contact hours" total. The 40 available hours are shown divided up in the table below.

KA	Topic	Hours
DES.ar	Architectural design	**15 Total**
DES.ar.1	Architectural styles, patterns, and frameworks	4
DES.ar.2	Architectural trade-offs among various attributes	5
DES.ar.3	Hardware and systems engineering issues in software architecture	1
DES.ar.4	Requirements traceability in architecture	1
DES.ar.5	Service-oriented architectures	2
DES.ar.6	Architectures for network, mobile, and embedded systems	1
DES.ar.7	Relationship between product architecture and structure of development organization, market	1
DES.hci	Human-computer interaction design	**4 Total**
DES.hci.9	Metaphors and conceptual models	4
VAV.tst	Testing	**8 Total**
VAV.tst.9	Testing across quality attributes (e.g. usability, security, compatibility, accessibility, performance etc.)	8
QUA	**Software Quality**	**8 Total**
QUA.pda.6	Assessment of product quality attributes (e.g. usability, reliability, availability, etc.)	8
SEC	**Security**	**5 Total**
SEC.dev	Developing secure software	1
SEC.dev.1	Building security into the software development life cycle	1
SEC.dev.2	Security in requirements analysis and specification	1
SEC.dev.3	Secure design principles and patterns	2

Additional topics

Students are expected to participate in course improvement. This means getting their feedback, and taking pre and post-course questionnaires regarding their level of understanding of course topics, among other things.

Other comments

(none)

B.9. Software Testing and Quality Assurance (SPSU)

SWE 3643 Software Testing and Quality Assurance
Southern Polytechnic State University (to be Kennesaw State Univ. in 2015)
Marietta, Georgia
Frank Tsui
ftsui@spsu.edu

http://cse.spsu.edu/ftsui (class notes available when I offer this course)

Catalogue description:

This course shows how to define software quality and how it is assessed through various testing techniques. Topics include review/inspection techniques for non-executable software, black-box and white-box testing techniques for executable software and test analysis. Specific test-case development techniques such as boundary value, equivalence class, control paths, and dataflow paths test are introduced. Different levels of testing such as functional, component, and system/regression tests are discussed with the concept of configuration management.

Expected Outcomes:

After taking this course, the student will be able to:

- Explore and understand the notion of quality and the definition of quality
- Understanding and setting quality goals, measuring techniques, and analyzing product and process quality.
- Learn how to develop test plan, test process, test scenarios, and test cases to achieve the quality goal.
- Exploring and mastering techniques to achieve the quality goals for software product through a) inspection/reviews, b) black/white box testing techniques, and c) verification using unit, component, system and regression test.
- Introduce the students to the notion of and techniques to achieve the quality goals for the software project through QA planning, through configuration management and through software development process improvement

Where does the course fit in your curriculum:

This is a 3-credit-hour required course taken by all undergraduate software engineering majors and game design majors in the second semester of their sophomore (2nd) year or later. Introduction to Software Engineering course is a pre-requisite for this course. Recent class size for this course has been approximately 30 to 35 students. Some computer science majors also take this course as an elective.

What is covered in the course:

Definitions, Basic Concept, and Relationships of Quality, Quality Assurance, and Testing
Overview of Different Testing Techniques
Testing of non-Executable: Inspection/Review Technique/Process (a la M. Fagan)
Review of Basic Sets and Propositional Calculus
Black Box (Functional)Testing techniques: Boundary Value testing, Equivalence Class based testing, Decision Table based testing, and their relationships
Review of Basic Graph Theory
White Box (Structural) Testing techniques: Path/Basis testing, Dataflow testing, Slice-based testing, and their relationships
Test Plan, Test Metrics and Test Tracking
Different Levels of Testing and Techniques for Unit testing, Functional testing, Integration testing, and System testing
Configuration management for Integration and System testing
Different models for Interaction Testing

What is the format of the course:

The course is taught in traditional face-to-face classroom style with lectures, student projects, and student presentations. The course meets for1.5 hours twice per week over a 16 -week semester (including final exam). Students also work on small teams outside of class a) to prepare for inspection/review which is conducted in class, b) to prepare for test case development, test execution, and test result documentation and analysis, c) to prepare for class presentation on product quality based on analysis of test goal, test team status, and test results.

How are students assessed?

Students are assessed individually through two closed book class-room exams. Students are also assessed by teams, based on their team projects in terms of their individual effort, contribution, and attitude. Team projects assessment also includes students' assessments of each other.

Course textbooks and materials:

There is one textbook:
Software Testing, A Craftsman's Approach, by Paul C. Jorgensen, Auerbach Publications, 2008 ISBN: 0-8493-7475-8
Additional readings are sometimes used for some topics (for example: "Advances in Software Inspections" by M. Fagan, "What is Software Testing and Why Is It So Hard" by J. Whittaker, "How to Design Practical Test Cases" by T. Yamaura, "Clearing a Career Path for Software Testers" by E. Weyuker , et al, etc.)

Pedagogical Advice:

Students tend to focus on various testing techniques and lose sight of why we are doing these tasks. So, they need to be reminded of why and how much different testing we need to perform in relationship to various levels of quality goals.

Body of Knowledge coverage:

KA	**Knowledge Unit**	**Hours**
QUA.pda QUA.pca VAV.fnd	Basic definitions, concepts, and relationships among quality, quality assurance (product and process), and testing.	3.0 hours
VAV.fnd	Introductory definitions and concepts of different testing techniques (for non-executables and executables), test process, and levels of testing	1.5
VAV.rev	Inspection and review techniques and process for non-executables such as requirements and design documents	3.0
FND.mf	Basic set theory and propositional calculus for testing	1.5
VAV.fnd VAV.tst	General Concept of Black- Box (functional testing techniques) and Boundary-Value/Robustness testing	4.0
VAV.tst	Equivalence class based testing technique	1.5
VAV.tst	Decision-table based testing technique	1.5
FND.ef	Basic graph theory, paths, and adjacency matrix for testing	1.5

VAV.fnd VAV.tst	General Concept of White-box (structural testing technique) and various paths-based coverage testing techniques, including Basis testing and Cyclomatic complexity number	5.0
VAV.tst	Dataflow testing	3.0
VAV.tst	Slice-based testing	1.0
VAV.par	Evaluation of and metrics for relationship of gaps and redundancies among the different Structural Testing techniques	1.5
PRO.pp VAV.par QUA.pca	Test planning, test metrics, and test status tracking process and techniques	3.0
VAV.par PRO.cm	Test Execution Processes, Levels of Testing and Control, and Configuration Management	2.0
VAV.tst	Integration testing techniques (top down, bottom-up, neighborhood, MM-path, etc.) and metrics	2.0
VAV.rev VAV.tst	Systems and Regression testing techniques using threads and operational profile; relationship to customer "acceptance" test	2.0
FND.ef VAV.rev VAV.tst	Interaction testing and modelling techniques using petri-net, state machine, decision tables, object oriented classes, etc.	4.5

Additional topics
(none)
Other comments
(none)