Software Engineering (SE)

In every computing application domain, professionalism, quality, schedule, and cost are critical to producing software systems. Because of this, the elements of software engineering are applicable to developing software in all areas of computing. A wide variety of software engineering practices have been developed and utilized since the need for a discipline of software engineering was first recognized. Many trade-offs between these different practices have also been identified. Practicing software engineers have to select and apply appropriate techniques and practices to a given development effort to maximize value. To learn how to do this, they study the elements of software engineering.

Software engineering is the discipline concerned with the application of theory, knowledge, and practice to effectively and efficiently build reliable software systems that satisfy the requirements of customers and users. This discipline is applicable to small, medium, and large-scale systems. It encompasses all phases of the lifecycle of a software system, including requirements elicitation, analysis and specification; design; construction; verification and validation; deployment; and operation and maintenance. Whether small or large, following a traditional disciplined development process, an agile approach, or some other method, software engineering is concerned with the best way to build good software systems.

Software engineering uses engineering methods, processes, techniques, and measurements. It benefits from the use of tools for managing software development; analyzing and modeling software artifacts; assessing and controlling quality; and for ensuring a disciplined, controlled approach to software evolution and reuse. The software engineering toolbox has evolved over the years. For instance, the use of contracts, with requires and ensure clauses and class invariants, is one good practice that has become more common. Software development, which can involve an individual developer or a team or teams of developers, requires choosing the most appropriate tools, methods, and approaches for a given development environment.
Students and instructors need to understand the impacts of specialization on software engineering approaches. For example, specialized systems include:

- Real time systems
- Client-server systems
- Distributed systems
- Parallel systems
- Web-based systems
- High integrity systems
- Games
- Mobile computing
- Domain specific software (e.g., scientific computing or business applications)

Issues raised by each of these specialized systems demand specific treatments in each phase of software engineering. Students must become aware of the differences between general software engineering techniques and principles and the techniques and principles needed to address issues specific to specialized systems.

An important effect of specialization is that different choices of material may need to be made when teaching applications of software engineering, such as between different process models, different approaches to modeling systems, or different choices of techniques for carrying out any of the key activities. This is reflected in the assignment of core and elective material, with the core topics and learning outcomes focusing on the principles underlying the various choices, and the details of the various alternatives from which the choices have to be made being assigned to the elective material.

Another division of the practices of software engineering is between those concerned with the fundamental need to develop systems that implement correctly the functionality that is required for them, and those concerned with other qualities for systems and the trade-offs needed to balance these qualities. This division too is reflected in the assignment of core and elective material, so that topics and learning outcomes concerned with the basic methods for developing
such system are assigned to the core, and those that are concerned with other qualities and trade-offs between them are assigned to the elective material.

In general, students learn best at the application level much of the material defined in the SE KA by participating in a project. Such projects should require students to work on a team to develop a software system through as much of its lifecycle as is possible. Much of software engineering is devoted to effective communication among team members and stakeholders. Utilizing project teams, projects can be sufficiently challenging to require the use of effective software engineering techniques and that students develop and practice their communication skills. While organizing and running effective projects within the academic framework can be challenging, the best way to learn to apply software engineering theory and knowledge is in the practical environment of a project. The minimum hours specified for some knowledge units in this document may appear insufficient to accomplish associated application-level learning outcomes. It should be understood that these outcomes are to be achieved through project experience that may even occur later in the curriculum than when the topics within the knowledge unit are introduced.

Note: The SDF/Development Methods knowledge unit includes 10 Core-Tier1 hours that constitute an introduction to certain aspects of software engineering. The knowledge units, topics and core hour specifications in this document must be understood as assuming previous exposure to the material described in SDF/Development Methods.
SE. Software Engineering (6 Core-Tier1 hours; 21 Core-Tier2 hours)

<table>
<thead>
<tr>
<th>Course</th>
<th>Core-Tier1 hours</th>
<th>Core-Tier2 hours</th>
<th>Includes Electives</th>
</tr>
</thead>
<tbody>
<tr>
<td>SE/Software Processes</td>
<td>2</td>
<td>1</td>
<td>Y</td>
</tr>
<tr>
<td>SE/Software Project Management</td>
<td></td>
<td>2</td>
<td>Y</td>
</tr>
<tr>
<td>SE/Tools and Environments</td>
<td></td>
<td>2</td>
<td>N</td>
</tr>
<tr>
<td>SE/Requirements Engineering</td>
<td>1</td>
<td>3</td>
<td>Y</td>
</tr>
<tr>
<td>SE/Software Design</td>
<td>3</td>
<td>5</td>
<td>Y</td>
</tr>
<tr>
<td>SE/Software Construction</td>
<td></td>
<td>2</td>
<td>Y</td>
</tr>
<tr>
<td>SE/Software Verification and Validation</td>
<td></td>
<td>3</td>
<td>Y</td>
</tr>
<tr>
<td>SE/Software Evolution</td>
<td></td>
<td>2</td>
<td>Y</td>
</tr>
<tr>
<td>SE/Formal Methods</td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>SE/Software Reliability</td>
<td>1</td>
<td></td>
<td>Y</td>
</tr>
</tbody>
</table>

SE/Software Processes

[2 Core-Tier1 hours; 1 Core-Tier2 hour]

**Topics:**

- Systems level considerations, i.e., the interaction of software with its intended environment
- Introduction to software process models (e.g., waterfall, incremental, agile)
  - Phases of software life-cycles
- Programming in the large vs. individual programming

[Core-Tier2]

- Applying software process models

[Elective]

- Software quality concepts
- Process improvement
- Software process capability maturity models
- Software process measurements
Learning Outcomes:

1. Describe how software can interact with and participate in various systems including information management, embedded, process control, and communications systems. [Familiarity]
2. Describe the difference between principles of the waterfall model and models using iterations. [Familiarity]
3. Describe the different practices that are key components of various process model. [Familiarity]
4. Differentiate among the phases of software development. [Familiarity]
5. Describe how programming in the large differs from individual efforts with respect to understanding a large code base, code reading, understanding builds, and understanding context of changes. [Familiarity]

6. Explain the concept of a software life cycle and provide an example, illustrating its phases including the deliverables that are produced. [Familiarity]
7. Compare several common process models with respect to their value for development of particular classes of software systems taking into account issues such as requirement stability, size, and non-functional characteristics. [Usage]
8. Define software quality and describe the role of quality assurance activities in the software process. [Familiarity]
9. Describe the intent and fundamental similarities among process improvement approaches. [Familiarity]
10. Compare several process improvement models such as CMM, CMMI, CQI, Plan-Do-Check-Act, or ISO9000. [Familiarity]
11. Use a process improvement model such as PSP to assess a development effort and recommend approaches to improvement. [Usage]
12. Explain the role of process maturity models in process improvement. [Familiarity]
13. Describe several process metrics for assessing and controlling a project. [Familiarity]
14. Use project metrics to describe the current state of a project. [Usage]

SE/Software Project Management

[2 Core-Tier2 hours]

Topics:

1. Team participation
   • Team processes including responsibilities for tasks, meeting structure, and work schedule
   • Roles and responsibilities in a software team
   • Team conflict resolution
   • Risks associated with virtual teams (communication, perception, structure)
2. Effort Estimation (at the personal level)
3. Risk
   • The role of risk in the life cycle
   • Risk categories including security, safety, market, financial, technology, people, quality, structure
4. Team management
   • Team organization and decision-making

[Core-Tier2]
Role identification and assignment

Individual and team performance assessment

Project management
  - Scheduling and tracking
  - Project management tools
  - Cost/benefit analysis

Software measurement and estimation techniques

Software quality assurance and the role of measurements

Risk
  - Risk identification and management
  - Risk analysis and evaluation
  - Risk tolerance (e.g., risk-adverse, risk-neutral, risk-seeking)
  - Risk planning

System-wide approach to risk including hazards associated with tools

Learning Outcomes:

1. Identify behaviors that contribute to the effective functioning of a team. [Familiarity]
2. Create and follow an agenda for a team meeting. [Usage]
3. Identify and justify necessary roles in a software development team. [Usage]
4. Understand the sources, hazards, and potential benefits of team conflict. [Usage]
5. Apply a conflict resolution strategy in a team setting. [Usage]
6. Use an ad hoc method to estimate software development effort (e.g., time) and compare to actual effort required. [Usage]
7. List several examples of software risks. [Familiarity]
8. Describe the impact of risk in a software development life cycle. [Familiarity]
9. Describe different categories of risk in software systems. [Familiarity]
10. Identify security risks for a software system. [Usage]
11. Demonstrate through involvement in a team project the central elements of team building and team management. [Usage]
12. Identify several possible team organizational structures and team decision-making processes. [Familiarity]
13. Create a team by identifying appropriate roles and assigning roles to team members. [Usage]
14. Assess and provide feedback to teams and individuals on their performance in a team setting. [Usage]
15. Prepare a project plan for a software project that includes estimates of size and effort, a schedule, resource allocation, configuration control, change management, and project risk identification and management. [Usage]
16. Track the progress of a project using appropriate project metrics. [Usage]
17. Compare simple software size and cost estimation techniques. [Usage]
18. Use a project management tool to assist in the assignment and tracking of tasks in a software development project. [Usage]
19. Describe the impact of risk tolerance on the software development process. [Assessment]
20. Identify risks and describe approaches to managing risk (avoidance, acceptance, transference, mitigation), and characterize the strengths and shortcomings of each. [Familiarity]
21. Explain how risk affects decisions in the software development process. [Usage]
22. Demonstrate a systematic approach to the task of identifying hazards and risks in a particular situation. [Usage]
23. Apply the basic principles of risk management in a variety of simple scenarios including a security situation. [Usage]
24. Conduct a cost/benefit analysis for a risk mitigation approach. [Usage]
25. Identify and analyze some of the risks for an entire system that arise from aspects other than the software.

[Usage]

SE/Tools and Environments

[2 Core-Tier2 hours]

Topics:

[Core-Tier2]

- Software configuration management and version control; release management
- Requirements analysis and design modeling tools
- Testing tools including static and dynamic analysis tools
- Programming environments that automate parts of program construction processes (e.g., automated builds)
  - Continuous integration
- Tool integration concepts and mechanisms

Learning Outcomes:

[Core-Tier2]

1. Describe the difference between centralized and distributed software configuration management.
   [Familiarity]
2. Identify configuration items and use a source code control tool in a small team-based project. [Usage]
3. Describe the issues that are important in selecting a set of tools for the development of a particular software system, including tools for requirements tracking, design modeling, implementation, build automation, and testing. [Familiarity]
4. Demonstrate the capability to use software tools in support of the development of a software product of medium size. [Usage]

SE/Requirements Engineering

[1 Core-Tier1 hour; 3 Core-Tier2 hours]

Topics:

[Core-Tier1]

- Properties of requirements including consistency, validity, completeness, and feasibility
- Describing functional requirements using, for example, use cases or users stories

[Core-Tier2]

- Software requirements elicitation
- Non-functional requirements and their relationship to software quality
- Describing system data using, for example, class diagrams or entity-relationship diagrams
- Evaluation and use of requirements specifications

[Elective]

- Requirements analysis modeling techniques
- Acceptability of certainty / uncertainty considerations regarding software / system behavior
- Prototyping
- Basic concepts of formal requirements specification
- Requirements specification
- Requirements validation
- Requirements tracing

**Learning Outcomes:**

[Core-Tier1]

1. List the key components of a use case or similar description of some behavior that is required for a system and discuss their role in the requirements engineering process. [Familiarity]
2. Interpret a given requirements model for a simple software system. [Familiarity]
3. Conduct a review of a set of software requirements to determine the quality of the requirements with respect to the characteristics of good requirements. [Usage]

[Core-Tier2]

4. Describe the fundamental challenges of and common techniques used for requirements elicitation. [Familiarity]
5. List the key components of a class diagram or similar description of the data that a system is required to handle. [Familiarity]
6. Identify both functional and non-functional requirements in a given requirements specification for a software system. [Usage]

[Elective]

7. Apply key elements and common methods for elicitation and analysis to produce a set of software requirements for a medium-sized software system. [Usage]
8. Use a common, non-formal method to model and specify (in the form of a requirements specification document) the requirements for a medium-size software system [Usage]
9. Translate into natural language a software requirements specification (e.g., a software component contract) written in a formal specification language. [Usage]
10. Create a prototype of a software system to mitigate risk in requirements. [Usage]
11. Differentiate between forward and backward tracing and explain their roles in the requirements validation process. [Familiarity]

**SE/Software Design**

[3 Core-Tier1 hours; 5 Core-Tier2 hours]

**Topics:**

[Core-Tier1]

- Overview of design paradigms
- System design principles: divide and conquer (architectural design and detailed design), separation of concerns, information hiding, coupling and cohesion, re-use of standard structures.
- Appropriate models of software designs, including structure and behavior.

[Core-Tier2]

- Design Paradigms such as structured design (top-down functional decomposition), object-oriented analysis and design, event driven design, component-level design, data-structured centered, aspect oriented, function oriented, service oriented.
- Relationships between requirements and designs: transformation of models, design of contracts, invariants.
• Software architecture concepts and standard architectures (e.g. client-server, n-layer, transform centered, pipes-and-filters, etc).
• Refactoring designs and the use of design patterns.
• The use of components in design: component selection, design, adaptation and assembly of components, components and patterns, components and objects, (for example, build a GUI using a standard widget set).
• [Elective]
  • Internal design qualities, and models for them: efficiency and performance, redundancy and fault tolerance, traceability of requirements.
  • External design qualities, and models for them: functionality, reliability, performance and efficiency, usability, maintainability, portability.
  • Measurement and analysis of design quality.
  • Tradeoffs between different aspects of quality.
  • Application frameworks.
  • Middleware: the object-oriented paradigm within middleware, object request brokers and marshalling, transaction processing monitors, workflow systems.

Learning Outcomes:

[Core-Tier1]

1. Articulate design principles including separation of concerns, information hiding, coupling and cohesion, and encapsulation. [Familiarity]
2. Use a design paradigm to design a simple software system, and explain how system design principles have been applied in this design. [Usage]
3. Construct models of the design of a simple software system that are appropriate for the paradigm used to design it. [Usage]
4. For the design of a simple software system within the context of a single design paradigm, describe the software architecture of that system. [Familiarity]
5. Within the context of a single design paradigm, describe one or more design patterns that could be applicable to the design of a simple software system. [Familiarity]

[Core-Tier2]

6. For a simple system suitable for a given scenario, discuss and select an appropriate design paradigm. [Usage]
7. Create appropriate models for the structure and behavior of software products from their requirements specifications. [Usage]
8. Explain the relationships between the requirements for a software product and the designed structure and behavior, in terms of the appropriate models and transformations of them. [Assessment]
9. Apply simple examples of patterns in a software design. [Usage]
10. Given a high-level design, identify the software architecture by differentiating among common software architectures such as 3-tier, pipe-and-filter, and client-server. [Familiarity]
11. Investigate the impact of software architectures selection on the design of a simple system. [Assessment]
12. Select suitable components for use in the design of a software product. [Usage]
13. Explain how suitable components might need to be adapted for use in the design of a software product. [Familiarity].
14. Design a contract for a typical small software component for use in a given system. [Usage]

[Elective]

15. Discuss and select appropriate software architecture for a simple system suitable for a given scenario. [Usage]
16. Apply models for internal and external qualities in designing software components to achieve an acceptable tradeoff between conflicting quality aspects. [Usage]
17. Analyze a software design from the perspective of a significant internal quality attribute. [Assessment]
18. Analyze a software design from the perspective of a significant external quality attribute. [Assessment]
19. Explain the role of objects in middleware systems and the relationship with components. [Familiarity]
20. Apply component-oriented approaches to the design of a range of software, such as using components for services for remote query and database management, or for secure communication and access. [Usage]

SE/Software Construction

[2 Core-Tier2 hours]

Topics:

[Core-Tier2]

- Coding practices: techniques, idioms/patterns, mechanisms for building quality programs
  - Defensive coding practices
  - Secure coding practices
  - Using exception handling mechanisms to make programs more robust, fault-tolerant
- Coding standards
- Integration strategies
- Development context: “green field” vs. existing code base
  - Change impact analysis
  - Change actualization

[Elective]

- Robust And Security Enhanced Programming
  - Defensive programming
  - Principles of secure design and coding:
    - Principle of least privilege
    - Principle of fail-safe defaults
    - Principle of psychological acceptability
- Potential security problems in programs
  - Buffer and other types of overflows
  - Race conditions
  - Improper initialization, including choice of privileges
  - Checking input
  - Assuming success and correctness
  - Validating assumptions
- Documenting security considerations in using a program

Learning Outcomes:

[Core-Tier2]

1. Describe techniques, coding idioms and mechanisms for implementing designs to achieve desired properties such as reliability, efficiency, and robustness. [Familiarity]
2. Build robust code using exception handling mechanisms. [Usage]
3. Describe secure coding and defensive coding practices. [Familiarity]
4. Select and use a defined coding standard in a small software project. [Usage]
5. Compare and contrast integration strategies including top-down, bottom-up, and sandwich integration. [Familiarity]

6. Describe the process of analyzing and implementing changes to code base developed for a specific project. [Familiarity]

7. Describe the process of analyzing and implementing changes to a large existing code base. [Familiarity]

[Elective]

8. Rewrite a simple program to remove common vulnerabilities, such as buffer overflows, integer overflows and race conditions [Usage]

9. State and apply the principles of least privilege and fail-safe defaults. [Familiarity]

10. Write a simple library that performs some non-trivial task and will not terminate the calling program regardless of how it is called [Usage]

SE/Software Verification Validation
[3 Core-Tier2 hours]

Topics:

[Core-Tier2]

- Verification and validation concepts
- Inspections, reviews, audits
- Testing types, including human computer interface, usability, reliability, security, conformance to specification
- Testing fundamentals
  - Unit, integration, validation, and system testing
  - Test plan creation and test case generation
  - Black-box and white-box testing techniques
- Defect tracking
- Testing parallel and distributed systems

[Elective]

- Static approaches and dynamic approaches to verification
- Regression testing
- Test-driven development
- Validation planning; documentation for validation
- Object-oriented testing; systems testing
- Verification and validation of non-code artifacts (documentation, help files, training materials)
- Fault logging, fault tracking and technical support for such activities
- Fault estimation and testing termination including defect seeding

Learning Outcomes:

[Core-Tier2]

1. Distinguish between program validation and verification. [Familiarity]

2. Describe the role that tools can play in the validation of software. [Familiarity]

3. Undertake, as part of a team activity, an inspection of a medium-size code segment. [Usage]

4. Describe and distinguish among the different types and levels of testing (unit, integration, systems, and acceptance). [Familiarity]
5. Describe techniques for identifying significant test cases for unit, integration, and system testing. [Familiarity]

6. Use a defect tracking tool to manage software defects in a small software project. [Usage]

7. Describe the issues and approaches to testing parallel and distributed systems. [Familiarity]

[Elective]

8. Create, evaluate, and implement a test plan for a medium-size code segment. [Usage]

9. Compare static and dynamic approaches to verification. [Familiarity]

10. Discuss the issues involving the testing of object-oriented software. [Usage]

11. Describe techniques for the verification and validation of non-code artifacts. [Familiarity]

12. Describe approaches for fault estimation. [Familiarity]

13. Estimate the number of faults in a small software application based on fault density and fault seeding. [Usage]

14. Conduct an inspection or review of software source code for a small or medium sized software project. [Usage]

**SE/Software Evolution**

*[2 Core-Tier2 hour]*

**Topics:**

- Software development in the context of large, pre-existing code bases
  - Software change
  - Concerns and concern location
  - Refactoring
- Software evolution
- Characteristics of maintainable software
- Reengineering systems
- Software reuse

**Learning Outcomes:**

1. Identify the principal issues associated with software evolution and explain their impact on the software life cycle. [Familiarity]

2. Estimate the impact of a change request to an existing product of medium size. [Usage]

3. Identify weaknesses in a given simple design, and removed them through refactoring. [Usage]

4. Discuss the challenges of evolving systems in a changing environment. [Familiarity]

5. Outline the process of regression testing and its role in release management. [Usage]

6. Discuss the advantages and disadvantages of software reuse. [Familiarity]

**SE/Formal Methods**

*[Elective]*
The topics listed below have a strong dependency on core material from the Discrete Structures area, particularly knowledge units DS/Functions Relations And Sets, DS/Basic Logic and DS/Proof Techniques.

Topics:

- Role of formal specification and analysis techniques in the software development cycle
- Program assertion languages and analysis approaches (including languages for writing and analyzing pre- and post-conditions, such as OCL, JML)
- Formal approaches to software modeling and analysis
  - Model checkers
  - Model finders
- Tools in support of formal methods

Learning Outcomes:

1. Describe the role formal specification and analysis techniques can play in the development of complex software and compare their use as validation and verification techniques with testing. [Familiarity]
2. Apply formal specification and analysis techniques to software designs and programs with low complexity. [Usage]
3. Explain the potential benefits and drawbacks of using formal specification languages. [Familiarity]
4. Create and evaluate program assertions for a variety of behaviors ranging from simple through complex. [Usage]
5. Using a common formal specification language, formulate the specification of a simple software system and derive examples of test cases from the specification. [Usage]

SE/Software Reliability

[1 Core-Tier2]

Topics:

[Core-Tier2]

- Software reliability engineering concepts
- Software reliability, system reliability and failure behavior (cross-reference SF9/Reliability Through Redundancy)
- Fault lifecycle concepts and techniques

[Elective]

- Software reliability models
- Software fault tolerance techniques and models
- Software reliability engineering practices
- Measurement-based analysis of software reliability

Learning Outcomes:

[Core-Tier2]

1. Explain the problems that exist in achieving very high levels of reliability. [Familiarity]
2. Describe how software reliability contributes to system reliability [Familiarity]
3. List approaches to minimizing faults that can be applied at each stage of the software lifecycle. [Familiarity]

4. Compare the characteristics of three different reliability modeling approaches. [Familiarity]

5. Demonstrate the ability to apply multiple methods to develop reliability estimates for a software system. [Usage]

6. Identify methods that will lead to the realization of a software architecture that achieves a specified reliability level of reliability. [Usage]

7. Identify ways to apply redundancy to achieve fault tolerance for a medium-sized application. [Usage]