Software Engineering (SE)

Preamble

As far back as the early 1970s, Brian Randell allegedly said, “Software engineering is the multi-person construction of multi-version programs.” This is an essential insight: while programming is the skill that governs our ability to write a program, software engineering is distinct in two dimensions: time and people.

First, a software engineering project is a team endeavor. Being a solitary programming expert is insufficient. Skilled software engineers will additionally demonstrate expertise in communication and collaboration. Programming may be an individual activity, but software engineering is a collaborative one, deeply tied to issues of professionalism, teamwork, and communication.

Second, a software engineering project is usually “multi-version.” It has an expected lifespan; it needs to function properly for months, years, or decades. Features may be added or removed to meet product requirements. The technological context will change, as our computing platforms evolve, programming languages change, dependencies upgrade, etc. This exposure to matters of time and change is novel when compared to a programming project: it isn’t enough to build a thing that works, instead it must work and stay working. Many of the most challenging topics in tech share “time will lead to change” as a root cause: backward compatibility, version skew, dependency management, schema changes, protocol evolution.

Software engineering presents a particularly difficult challenge for learning in an academic setting. Given that the major differences between programming and Software engineering are time and teamwork, it is hard to generate lessons that *require* successful teamwork and that faithfully present the risks of time. Additionally, some topics in software engineering will be more authentic and more relevant if and when our learners experience collaborative and long-term software engineering projects *in vivo* rather than in the classroom. Regardless of whether that happens as an internship, involvement in an open source project, or full-time engineering role, a month of full-time hands-on experience has more available hours than the average software engineering course.

Thus, a software engineering curriculum should focus primarily on ideas that are needed by a majority of new-grad hires, and that either are novel for those who are trained primarily as programmers, or that are abstract concepts that may not get explicitly stated/shared on the job. Such topics include, but are not limited to:

- Testing
- Teamwork, collaboration
- Communication
- Design
- Maintenance and Evolution

- Software engineering tools

Some such material is reasonably suited to a standard lecture or lecture+lab course. Discussing theoretical underpinnings of version control systems, or branching strategies in such systems, can be an effective way to familiarize students with those ideas. Similarly, a theoretical discussion can highlight the difference between static and dynamic analysis tools, or may motivate discussion of diamond dependency problems in dependency networks.

On the other hand, many of the fundamental topics of software engineering are best experienced in a hands-on fashion. Historically, project-oriented courses have been a common vehicle for such learning. We believe that such experience is valuable but also bears some interesting risks: students may form erroneous notions about the difficulty / complexity of collaboration if their only exposure is a single project with teams formed of other novice software engineers. It falls to instructors to decide on the right balance between theoretical material and hands-on projects - neither is a perfect vehicle for this challenging material. We strongly encourage instructors of project courses to aim for iteration and fast feedback - a few simple tasks repeated (i.e., in an Agile-structured project) is better than singular high-friction introductions to many types of tasks. Programs with real-world industry partners and clients are also particularly encouraged. If long-running project courses are not an option, anything that can expose learners to the collaborative and long-term aspects of software engineering is valuable: adding features to an existing codebase, collaborating on distinct parts of a larger whole, pairing up to write an encoder and decoder, etc.

All evidence suggests that the role of software in our society will continue to grow for the foreseeable future, and yet the era of “two programmers in a garage” seems to have drawn to a close. Most important software these days is clearly a team effort, building on existing code and leveraging existing functionality. The study of software engineering skills is a deeply important counterpoint to the everyday experience of computing students - we *must* impress on them the reality that few software projects are managed by writing from scratch as a solo endeavor. Communication, teamwork, planning, testing, and tooling are far more important as our students move on from the classroom and make their mark on the wider world.

Changes since CS 2013

This document shifts the focus of the Software Engineering knowledge area in a few ways compared to the goals of CS2013. The common reasoning behind most of these changes is to focus on material that learners would not pick up elsewhere in the curriculum, and that will be relevant *immediately* upon graduation, rather than at some future point in their careers.

- More explicit focus on the software workflow (version control, testing, code review, tooling)
- Less focus on team *leadership* and project management.
- More focus on team *participation*, communication, and collaboration

Overview

SE-Teamwork: Because of the nature of learning programming, most students in introductory SE have little or no exposure to the collaborative nature of SE. Practice (for instance in project work) may help, but lecture and discussion time spent on the value of clear, effective, and efficient communication and collaboration. are essential for Software Engineering.

SE-Tools: Industry reliance on SE tools has exploded in the past generation, with version control becoming ubiquitous, testing frameworks growing in popularity, increased reliance on static and dynamic analysis in practice, and near-ubiquitous use of continuous integration systems. Increasingly powerful IDEs provide code searching and indexing capabilities, as well as small scale refactoring tools and integration with other SE tools. An understanding of the nature of these tools is broadly valuable - especially version control systems.

SE-Requirements: Knowing how to build something is of little help if we do not know what to build. Product Requirements (aka Requirements Engineering, Product Design, Product Requirements solicitation, PRDs, etc.) introduces students to the processes surrounding the specification of the broad requirements governing development of a new product or feature.

SE-Design: While Product Requirements focuses on the user-facing functionality of a software system, Software Design focuses on the engineer-facing design of internal software components. This encompasses large design concerns such as software architecture, as well as small-scale design choices like API design.

SE-Construction: Software Construction focuses on practices that influence the direct production of software: use of tests, test driven development, coding style. More advanced topics extend into secure coding, dependency injection, work prioritization, etc.

SE-Validation: Software Verification and Validation focuses on how to improve the value of testing - understand the role of testing, failure modes, and differences between good tests and poor ones.

SE-Refactoring: Refactoring and Code Evolution focuses on refactoring and maintenance strategies, incorporating code health, use of tools, and backwards compatibility considerations.

SE-Reliability: Software Reliability aims to improve understanding of and attention to error cases, failure modes, redundancy, and reasoning about fault tolerance.

SE-FormalMethods: Formal Methods provides mathematically rigorous mechanisms to apply to software, from specification to verification. (Prerequisites: Substantial dependence on core material from the Discrete Structures area, particularly knowledge units DS/Basic Logic and DS/Proof Techniques.)

Core Hours

Knowledge Units	**CS Core**	**KA Core**
Teamwork	2	2
Tools and Environments	1	3
Product Requirements		2
Software Design	1	4
Software Construction	1	3
Software Verification and Validation	1	3
Refactoring and Code Evolution		2
Software Reliability		2
Formal Methods		
Total	**6**	**21**

Knowledge Units

SE-A: Teamwork (SE-Teamwork)

CS Core:

1. Effective communication, including oral and written, as well as formal (email, docs, comments, presentations) and informal (team chat, meetings) (See also: SEP/Professional Communication)
2. Common causes of team conflict, and approaches for conflict resolution
3. Cooperative programming
    a. Pair programming or Swarming
    b. Code review
    c. Collaboration through version control
4. Roles and responsibilities in a software team (See also: SEP/Professional Ethics)
    a. Advantages of teamwork
    b. Risks and complexity of such collaboration
5. Team processes
    a. Responsibilities for tasks, effort estimation, meeting structure, work schedule
6. Importance of team diversity and inclusivity (See also: SEP/Professional Communication)

KA Core:

7. Interfacing with stakeholders, as a team
    a. Management & other non-technical teams
    b. Customers
    c. Users
8. Risks associated with physical, distributed, hybrid and virtual teams
    a. Including communication, perception, structure, points of failure, mitigation and recovery, etc.

Illustrative Learning Outcomes:

CS Core:

1. Follow effective team communication practices.
2. Articulate the sources of, hazards of, and potential benefits of team conflict - especially focusing on the value of disagreeing about ideas or proposals without insulting people.
3. Facilitate a conflict resolution and problem solving strategy in a team setting.
4. Collaborate effectively in cooperative development/programming.
5. Propose and delegate necessary roles and responsibilities in a software development team.
6. Compose and follow an agenda for a team meeting.
7. Facilitate through involvement in a team project, the central elements of team building, establishing healthy team culture, and team management including creating and executing a team work plan.
8. Promote the importance of and benefits that diversity and inclusivity brings to a software development team

KA Core:

9. Reference the importance of, and strategies to, as a team, interface with stakeholders outside the team on both technical and non-technical levels.
10. Enumerate the risks associated with physical, distributed, hybrid and virtual teams and possible points of failure and how to mitigate against and recover/learn from failures.

SE-B: Tools and Environments (SE-Tools)

CS Core:

1. Software configuration management and version control (See also: SDF/Software Development)
    a. Configuration in version control, reproducible builds/configuration
    b. Version control branching strategies. Development branches vs. release branches. Trunk-based development.
    c. Merging/rebasing strategies, when relevant.

KA Core:

2. Release management
3. Testing tools including static and dynamic analysis tools (See also: SDF/Software Development)
4. Software process automation
    a. Build systems - the value of fast, hermetic, reproducible builds, compare/contrast approaches to building a project
    b. Continuous Integration (CI) - the use of automation and automated tests to do preliminary validation that the current head/trunk revision builds and passes (basic) tests
    c. Dependency management - updating external/upstream dependencies, package management, SemVer
5. Design and communication tools (docs, diagrams, common forms of design diagrams like UML)
6. Tool integration concepts and mechanisms (See also: SDF/Software Development)
7. Use of modern IDE facilities - debugging, refactoring, searching/indexing, ML-powered code assistants, etc. (See also: SDF/Software Development)

Illustrative Learning Outcomes:

CS Core:

1. Describe the difference between centralized and distributed software configuration management.
2. Describe how version control can be used to help manage software release management.
3. Identify configuration items and use a source code control tool in a small team-based project.

KA Core:

4. Describe how available static and dynamic test tools can be integrated into the software development environment.
5. Understand the use of CI systems as a ground-truth for the state of the team's shared code (build and test success).
6. 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.
7. Demonstrate the capability to use software tools in support of the development of a software product of medium size.

SE-C: Product Requirements (SE-Requirements)

KA Core:

1. Describe functional requirements using, for example, use cases or user stories
   a. Using at least one method of documenting and structuring functional requirements
   b. Understanding how the method supports design and implementation
   c. Strengths and weaknesses of using a particular approach
2. Properties of requirements including consistency, validity, completeness, and feasibility
3. Requirements elicitation
   a. Sources of requirements, for example, users, administrators, or support personnel
   b. Methods of requirement gathering, for example, surveys, interviews, or behavioral analysis
4. Non-functional requirements, for example, security, usability, or performance (aka Quality Attributes)
5. Risk identification and management
6. Communicating and/or formalizing requirement specifications

Non-core:

7. Prototyping
   a. A tool for both eliciting and validating/confirming requirements
8. Product evolution
   a. When requirements change, how to understand what effect that has and what changes need to be made
9. Effort estimation
   a. Learning techniques for better estimating the effort required to complete a task
   b. Practicing estimation and comparing to how long tasks actually take
   c. Effort estimation is quite difficult, so students are likely to be way off in many cases, but seeing the process play out with their own work is valuable

Illustrative Learning Outcomes:

KA Core:

1. Compare different methods of eliciting requirements along multiple axes.
2. Identify differences between two methods of describing functional requirements (e.g., customer interviews, user studies, etc.) and the situations where each would be preferred.
3. Identify which behaviors are required, allowed, or barred from a given set of requirements and a list of candidate behaviors.
4. Collect a set of requirements for a simple software system.
5. Identify areas of a software system that need to be changed, given a description of the system and a set of new requirements to be implemented.
6. Identify the functional and non-functional requirements in a set of requirements.

Non-core:

7. Create a prototype of a software system to validate a set of requirements. (Building a mock-up, MVP, etc.)
8. Estimate the time to complete a set of tasks, then compare estimates to the actual time taken.
9. Determine an implementation sequence for a set of tasks, adhering to dependencies between them, with a goal to retire risk as early as possible.
10. Write a requirement specification for a simple software system.

SE-D: Software Design (SE-Design)

CS Core:

1. System design principles (See also: SF/System Reliability)
    a. Levels of abstraction (e.g., architectural design and detailed design)
    b. Separation of concerns
    c. Information hiding
    d. Coupling and cohesion
2. Software architecture (See also: SF/System Reliability)
    a. Design paradigms
        i. Top-down functional decomposition / layered design
        ii. Data-oriented architecture
        iii. Object-oriented analysis and design
        iv. Event-driven design
    b. Standard architectures (e.g., client-server and microservice architectures including REST discussions, n-layer, pipes-and-filters, Model View Controller)
    c. Identifying component boundaries and dependencies
3. Programming in the large vs. programming in the small (See also: SF/System Reliability)
4. Code smells and other indications of code quality, distinct from correctness.

KA Core:

5. API design principles
    a. Consistency
        i. Consistent APIs are easier to learn and less error-prone
        ii. Consistency is both internal (between different portions of the API) and external (following common API patterns)
    b. Composability

    c. Documenting contracts
        i. API operations should describe their effect on the system, but not generally their implementation
        ii. Preconditions, postconditions, and invariants
    d. Expandability
    e. Error reporting
        i. Errors should be clear, predictable, and actionable
        ii. Input that does not match the contract should produce an error
        iii. Errors that can be reliably managed without reporting should be managed
6. Identifying and codifying data invariants and time invariants
7. Structural and behavioral models of software designs
8. Data design (See also: IM/Data Modeling)
    a. Data structures
    b. Storage systems
9. Requirement traceability
    a. Understanding which requirements are satisfied by a design

Non-Core:

10. Design modeling, for instance with class diagrams, entity relationship diagrams, or sequence diagrams
11. Measurement and analysis of design quality
12. Principles of secure design and coding (See also: SEC/Security Analysis and Engineering)
    a. Principle of least privilege
    b. Principle of fail-safe defaults
    c. Principle of psychological acceptability
13. Evaluating design tradeoffs (e.g., efficiency vs. reliability, security vs. usability)

Illustrative Learning Outcomes:

CS Core:

1. Identify the standard software architecture of a given high-level design.
2. Select and use an appropriate design paradigm to design a simple software system and explain how system design principles have been applied in this design.
3. Adapt a flawed system design to better follow principles such as separation of concerns or information hiding.
4. Identify the dependencies among a set of software components in an architectural design.

KA Core:

5. Design an API for a single component of a large software system, including identifying and documenting each operation’s invariants, contract, and error conditions.
6. Evaluate an API description in terms of consistency, composability, and expandability.
7. Expand an existing design to include a new piece of functionality.
8. Design a set of data structures to implement a provided API surface.
9. Identify which requirements are satisfied by a provided software design.

Non-Core:

10. Translate a natural language software design into class diagrams.
11. Adapt a flawed system design to better follow the principles of least privilege and fail-safe defaults.

12. Contrast two software designs across different qualities, such as efficiency or usability.

SE-E: Software Construction (SE-Construction)

CS Core:

1. Practical small-scale testing (See also: SDF/Software Development Practices)
   a. Unit testing
   b. Test-driven development - This is particularly valuable for students psychologically, as it is far easier to engage constructively with the challenge of identifying challenging inputs for a given API (edge cases, corner cases) a priori. If they implement first, the instinct is often to avoid trying to crash their new creation, while a test-first approach gives them the intellectual satisfaction of spotting the problem cases and then watching as more tests pass during the development process.
2. Documentation (See also: SDF/Software Development Practices)
   a. Interface documentation - Describe interface requirements, potentially including (formal or informal) contracts, pre and post conditions, invariants.
   b. Implementation documentation should focus on tricky and non-obvious pieces of code, whether because the code is using advanced language features or the behavior of the code is complex. (Do not add comments that re-state common/obvious operations and simple language features.)
      i. Clarify dataflow, computation, etc., focusing on what the code is
      ii. Identify subtle/tricky pieces of code and refactor to be self-explanatory if possible, or provide appropriate comments to clarify.

KA Core:

3. Coding style (See also: SDF/Software Development Practices)
   a. Style guides
   b. Commenting
   c. Naming
4. "Best Practices" for coding: techniques, idioms/patterns, mechanisms for building quality programs (See also: SEC/Defensive Programming, SDF/Software Development Practices)
   a. Defensive coding practices
   b. Secure coding practices and principles
   c. Using exception handling mechanisms to make programs more robust, fault-tolerant
5. Debugging (See also: SDF/Software Development Practices)
6. Logging
7. Use of libraries and frameworks developed by others (See also: SDF/Software Development Practices)

Non-Core:

8. Larger-scale testing
   a. Test doubles (stubs, mocks, fakes)
   b. Dependency injection
9. Work sequencing, including dependency identification, milestones, and risk retirement
   a. Dependency identification: Identifying the dependencies between different tasks
   b. Milestones: A collection of tasks that serve as a marker of progress when completed. Ideally, the milestone encompasses a useful unit of functionality.
   c. Risk retirement: Identifying what elements of a project are risky and prioritizing completing tasks that address those risks

10. Potential security problems in programs (See also: SEC/Defensive Programming)
    a. Buffer and other types of overflows
    b. Race conditions
    c. Improper initialization, including choice of privileges
    d. Input validation
11. Documentation (autogenerated)
12. Development context: “green field” vs. existing code base
    a. Change impact analysis
    b. Change actualization
13. Release management
14. DevOps practices

Illustrative Learning Outcomes:

CS Core:

1. Write appropriate unit tests for a small component (several functions, a single type, etc.).
2. Write appropriate interface and (if needed) implementation comments for a small component.

KA Core:

3. Describe techniques, coding idioms and mechanisms for implementing designs to achieve desired properties such as reliability, efficiency, and robustness.
4. Write robust code using exception handling mechanisms.
5. Describe secure coding and defensive coding practices.
6. Select and use a defined coding standard in a small software project.
7. Compare and contrast integration strategies including top-down, bottom-up, and sandwich integration.
8. Describe the process of analyzing and implementing changes to code base developed for a specific project.
9. Describe the process of analyzing and implementing changes to a large existing code base.

Non-Core:

10. Rewrite a simple program to remove common vulnerabilities, such as buffer overflows, integer overflows and race conditions.
11. Write a software component that performs some non-trivial task and is resilient to input and run-time errors.

SE-F: Software Verification and Validation (SE-Validation)

CS Core:

1. Verification and validation concepts
    a. Verification: Are we building the thing right?
    b. Validation: Did we build the right thing?
2. Why testing matters
    a. Does the component remain functional as the code evolves?
3. Testing objectives
    a. Usability

    b. Reliability
    c. Conformance to specification
    d. Performance
    e. Security
4. Test kinds
    a. Unit
    b. Integration
    c. Validation
    d. System
5. Stylistic differences between tests and production code
    a. DAMP vs. DRY - more duplication is warranted in test code.

KA Core:

6. Test planning and generation
    a. Test case generation, from formal models, specifications, etc.
    b. Test coverage
        i. Test matrices
        ii. Code coverage (how much of the code is tested)
        iii. Environment coverage (how many hardware architectures, OSes, browsers, etc. are tested)
    c. Test data and inputs
7. Test development
    a. Test-driven development
    b. Object oriented testing, mocking, and dependency injection
    c. Black-box and white-box testing techniques
    d. Test tooling, including code coverage, static analysis, and fuzzing
8. Verification and validation in the development cycle
    a. Code reviews
    b. Test automation, including automation of tooling
    c. Pre-commit and post-commit testing
    d. Trade-offs between test coverage and throughput/latency of testing
    e. Defect tracking and prioritization
        i. Reproducibility of reported defects
9. Domain specific verification and validation challenges
    a. Performance testing and benchmarking
    b. Asynchrony, parallelism, and concurrency
    c. Safety-critical
    d. Numeric

Non-Core:

10. Verification and validation tooling and automation
    a. Static analysis
    b. Code coverage
    c. Fuzzing
    d. Dynamic analysis and fault containment (sanitizers, etc.)
    e. Fault logging and fault tracking
11. Test planning and generation
    a. Fault estimation and testing termination including defect seeding
    b. Use of random and pseudo random numbers in testing
12. Performance testing and benchmarking
    a. Throughput and latency

   b. Degradation under load (stress testing, FIFO vs. LIFO handling of requests)
   c. Speedup and scaling
      i. Amadhl's law
      ii. Gustafson's law
      iii. Soft and weak scaling
   d. Identifying and measuring figures of merits
   e. Common performance bottlenecks
      i. Compute-bound
      ii. Memory-bandwidth bound
      iii. Latency-bound
   f. Statistical methods and best practices for benchmarking
      i. Estimation of uncertainty
      ii. Confidence intervals
   g. Analysis and presentation (graphs, etc.)
   h. Timing techniques
13. Testing asynchronous, parallel, and concurrent systems
14. Verification and validation of non-code artifacts (documentation, training materials)

Illustrative Learning Outcomes:

CS Core:

1. Explain why testing is important.
2. Distinguish between program validation and verification.
3. Describe different objectives of testing.
4. Compare and contrast the different types and levels of testing (regression, unit, integration, systems, and acceptance).

KA Core:

5. Describe techniques for creating a test plan and generating test cases.
6. Create a test plan for a medium-size code segment which includes a test matrix and generation of test data and inputs.
7. Implement a test plan for a medium-size code segment.
8. Identify the fundamental principles of test-driven development methods and explain the role of automated testing in these methods.
9. Discuss issues involving the testing of object-oriented software.
10. Describe mocking and dependency injection and their application.
11. Undertake, as part of a team activity, a code review of a medium-size code segment.
12. Describe the role that tools can play in the validation of software.
13. Automate testing in a small software project.
14. Explain the roles, pros, and cons of pre-commit and post-commit testing.
15. Discuss the tradeoffs between test coverage and test throughput/latency and how this can impact verification.
16. Use a defect tracking tool to manage software defects in a small software project.
17. Discuss the limitations of testing in certain domains.

Non-Core:

18. Describe and compare different tools for verification and validation.
19. Automate the use of different tools in a small software project.
20. Explain how and when random numbers should be used in testing.

21. Describe approaches for fault estimation.
22. Estimate the number of faults in a small software application based on fault density and fault seeding.
23. Describe throughput and latency and provide examples of each.
24. Explain speedup and the different forms of scaling and how they are computed.
25. Describe common performance bottlenecks.
26. Describe statistical methods and best practices for benchmarking software.
27. Explain techniques for and challenges with measuring time when constructing a benchmark.
28. Identify the figures of merit, construct and run a benchmark, and statistically analyze and visualize the results for a small software project.
29. Describe techniques and issues with testing asynchronous, concurrent, and parallel software.
30. Create a test plan for a medium-size code segment which contains asynchronous, concurrent, and/or parallel code, including a test matrix and generation of test data and inputs.
31. Describe techniques for the verification and validation of non-code artifacts.

SE-G: Refactoring and Code Evolution (SE-Refactoring)

KA Core:

1. Hyrum's Law / The Law of Implicit Interfaces
2. Backward compatibility
    a. Compatibility is not a property of a single entity, it's a property of a *relationship*.
    b. Backward compatibility needs to be evaluated in terms of provider + consumer(s) or with a well-specified model of what forms of compatibility a provider aspires to / promises.
3. Refactoring
    a. Standard refactoring patterns (rename, inline, outline, etc.)
    b. Use of refactoring tools in IDE
    c. Application of static-analysis tools (to identify code in need of refactoring, generate changes, etc.)
    d. Value of refactoring as a remedy for technical debt
4. Versioning
    a. Semantic Versioning (SemVer)
    b. Trunk-based development

Non-Core:

5. "Large Scale" Refactoring - techniques when a refactoring change is too large to commit safely (large projects), or when it is impossible to synchronize change between provider + all consumers (multiple repositories, consumers with private code).
    a. Express both old and new APIs so that they can co-exist
    b. Minimize the size of *behavior* changes
    c. Why these techniques are required, (e.g., "API consumers I can see" vs "consumers I can't see")

Illustrative Learning Outcomes:
KA-Core:

1. Identify both explicit and implicit behavior of an interface, and identify potential risks from Hyrum's Law
2. Consider inputs from static analysis tools and/or Software Design principles to identify code in need of refactoring.

3. Identify changes that can be broadly considered “backward compatible,” potentially with explicit statements about what usage is or is not supported
4. Refactor the implementation of an interface to improve design, clarity, etc. with minimal/zero impact on existing users
5. Evaluate whether a proposed change is sufficiently safe given the versioning methodology in use for a given project

Non-Core:

6. Plan a complex multi-step refactoring to change default behavior of an API safely.

SE-H: Software Reliability (SE-Reliability)

KA Core:

1. Concept of reliability as probability of failure or mean time between failures, and faults as cause of failures
2. Identifying reliability requirements for different kinds of software
3. Software failures caused by defects/bugs, and so for high reliability goal is to have minimum defects - by injecting fewer defects (better training, education, planning), and by removing most of the injected defects (testing, code review, etc.)
4. Software reliability, system reliability and failure behavior
5. Defect injection and removal cycle, and different approaches for defect removal
6. Compare the “error budget” approach to reliability with the “error-free” approach, and identify domains where each is relevant

Non-Core:

7. Software reliability models
8. Software fault tolerance techniques and models
    a. Contextual differences in fault tolerance (e.g., crashing a flight critical system is strongly avoided, crashing a data processing system before corrupt data is written to storage is highly valuable)
9. Software reliability engineering practices - including reviews, testing, practical model checking
10. Identification of dependent and independent failure domains, and their impact on system reliability
11. Measurement-based analysis of software reliability - telemetry, monitoring and alerting, dashboards, release qualification metrics, etc.

Illustrative Learning Outcomes:

KA Core:

1. Describe how to determine the level of reliability required by a software system.
2. Explain the problems that exist in achieving very high levels of reliability.
3. Understand approaches to minimizing faults that can be applied at each stage of the software lifecycle.

Non-Core:

4. Demonstrate the ability to apply multiple methods to develop reliability estimates for a software system.
5. Identify methods that will lead to the realization of a software architecture that achieves a specified level of reliability.
6. Identify ways to apply redundancy to achieve fault tolerance.
7. Identify single-point-of-failure (SPF) dependencies in a system design.

SE-I: Formal Methods (SE-FormalMethods)

Non-Core:

1. Formal specification of interfaces
   a. Specification of pre- and post- conditions
   b. Formal languages for writing and analyzing pre- and post-conditions.
2. Problem areas well served by formal methods
   a. Lock-free programming, data races
   b. Asynchronous and distributed systems, deadlock, livelock, etc.
3. Comparison to other tools and techniques for defect detection
   a. Testing
   b. Fuzzing
4. Formal approaches to software modeling and analysis
   a. Model checkers
   b. Model finders

Illustrative 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.
2. Apply formal specification and analysis techniques to software designs and programs with low complexity.
3. Explain the potential benefits and drawbacks of using formal specification languages.

Professional Dispositions

- **Collaborative**: Software engineering is increasingly described as a “team sport” - successful software engineers are able to work with others effectively. Humility, respect, and trust underpin the collaborative relationships that are essential to success in this field.
- **Professional**: Software engineering produces technology that has the chance to influence literally billions of people. Awareness of our role in society, strong ethical behavior, and commitment to respectful day-to-day behavior outside of one’s team are essential.
- **Communicative:** No single software engineer on a project is likely to know all of the project details. Successful software projects depend on engineers communicating clearly and regularly in order to coordinate effectively.
- **Meticulous:** Software engineering requires attention to detail and consistent behavior from everyone on the team. Success in this field is clearly influenced by a meticulous

approach - comprehensive understanding, proper procedures, and a solid avoidance of cutting corners.
- **Accountable:** The collaborative aspects of software engineering also highlight the value of accountability. Failing to take responsibility, failing to follow through, and failing to keep others informed are all classic causes of team friction and bad project outcomes.

Math Requirements

Desirable:

- Introductory statistics (performance comparisons, evaluating experiments, interpreting survey results, etc.)

Course Packaging Suggestions

Advanced Course to include at least the following:
- SE-Teamwork: Teamwork - 4 hours
- SE-Tools: Tools and Environments - 4 hours
- SE-Requirements: Product Requirements - 2 hours
- SE-Design: Software Design - 5 hours
- SE-Construction: Software Construction - 4 hours
- SE-Validation: Verification and Validation - 4 hours
- SE-Refactoring: Refactoring and Code Evolution - 2 hours
- SE-Reliability: Software Reliability - 2 hours
- SEP-C: Professional Ethics and SEP-F: Professional Communication - 7 hours

Pre-requisites:
- SDF-A: Fundamental Programming Concepts and Practices

Skill statement: A student who completes this course should be able to perform good quality code review for colleagues (especially focusing on professional communication and teamwork needs), read and write unit tests, use basic software tools (IDEs, version control, static analysis tools) and perform basic activities expected of a new hire on a software team.

Committee

Chair: Titus Winters (Google, New York City, NY, USA)

Members:

- Brett A. Becker, University College Dublin, Dublin, Ireland
- Adam Vartanian, Cord, London, UK
- Bryce Adelstein Lelbach, NVIDIA, New York City, NY, USA
- Patrick Servello, CIWRO, Norman, OK, USA
- Pankaj Jalote, IIIT-Delhi, Delhi, India

- Christian Servin, El Paso Community College, El Paso, TX, USA

Contributors:

- Hyrum Wright, Google, Pittsburgh, PA, USA
- Olivier Giroux, Apple, Cupertino, CA, USA
- Gennadiy Civil, Google, New York City, NY, USA