The CMU Master of Software Engineering Core Curriculum

David Garlan
Alan Brown, Daniel Jackson
Jim Tomayko, Jeannette Wing

August 16, 1993
CMU-CS-93-180

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This report outlines the Core Curriculum of the Carnegie Mellon University Master of Software Engineering Program.

Keywords: MSE, software engineering, education, curriculum design

1 Introduction

Since 1989 Carnegie Mellon University has offered an academic program leading to a professional Master of Software Engineering (MSE) degree. The objective of the MSE program is to develop technical leaders in industrial software engineering. Students entering the program have at least two years industrial experience, and normally complete the program in sixteen months of full-time study.

The MSE program is centered on a *Software Development Studio*, supported by a *Core Curriculum* and a set of *elective tracks*. These components occupy approximately 40%, 30%, and 30% (respectively) of a student's time in the program.

In the Software Development Studio students plan and implement a significant software project for an external client over the full duration of the program. Students work in a team environment under the guidance of faculty advisors to analyze a problem, plan a software development project, execute a solution, and evaluate their work. The Studio is similar to the design studios that characterize architectural degree programs.

The MSE Core Curriculum develops foundational skills in the fundamentals of software engineering, with an emphasis on design, analysis, and management of large-scale software systems. The elective tracks provide an opportunity for students to develop deeper expertise in one of several specialties, such as real time systems, human-computer interfaces, or the organizational environment of software systems.

This report summarizes the current design of the Core Curriculum (as revised in Spring 1993). The reader is referred to other MSE documents for detailed descriptions of the program as a whole as well as the Studio and elective components of the program.

2 Synopsis of the Core Curriculum

The MSE Core Curriculum consists of the following five semester courses:

1. **Models of Software Systems:** Scientific foundations for software engineering depend on the use of precise, abstract models and logics for characterizing and reasoning about properties of software systems.

   Notes on Content: Emphasis on formal models. Specific notations not emphasized, although introduced for concreteness. Topics include state machines, algebraic models, process algebras, trace models, compositional mechanisms, abstraction relations, temporal logic. Examples are drawn from software applications.

2. **Methods of Software Development:** Practical development of software requires an understanding of successful methods for bridging the gap between a problem to be solved and a working software system.

   Notes on Content: Intent is to treat comprehensive "vertical" approaches to Requirements Analysis, Design, Creation, and Maintenance. Among the methods and notations that might be included are: OOA/OOD, JSD/JSP, VDM, Z, Larch, Structured Analysis/Design, Cleanroom, prototype-oriented development. Students should be able to use several of the design methods, understand scope of applicability of each method, know what kinds of automated support exists for the methods.

3. **Management of Software Development:** Large-scale software development requires the ability to manage resources – both human and computational – through control of the development process.

   Notes on Content: Considers management of both individual software development efforts and long-term capability improvement. Includes life cycle models, project management, process management, capability maturity models, product control (e.g., version and configuration management, change control), documentation standards, risk management, people management skills, organizational structures, product management, and requirements elicitation.

4. **Analysis of Software Artifacts:** To build, maintain, and reuse software systems effectively it is necessary to analyze the products of software development.

   Notes on Content: Includes both static and dynamic analyses: e.g., type checking, verification, testing, performance analysis, hazard analysis, reverse engineering, program slicing. Use of tools for analysis. Artifacts may include more than delivered code: specifications, designs, documentation, prototypes, test suites.

5. **Architectures of Software Systems:** Successful design of complex software systems requires ability to describe, evaluate, and create systems at an architectural level of abstraction.

   Notes on Content: Treats organization of complex software systems. Covers the "horizontal" design space associated with system structure and assignment of functionality to design components. Topics include common patterns of architectural design, tradeoff analysis at architectural level, domain-specific architectures, automated support for architectural design, and formal models of software architecture.

3 Phasing

The Management, Models, and Methods courses are offered in the Fall semester. The Analysis and Architecture courses are offered in the Spring semester. The *directive* is a required course that is used to balance the core content in an individualized way for each student. Typically the directive will be a course offered in the CMU environment that fills gaps in an entering student's background.

FALL 1	SPRING	SUMMER	FALL 2
Studio	*Studio*	*Studio*	*Studio*
Directive	*Electives*		
Models			*Electives*
Methods	**Analysis**		
Management	**Architectures**	*Elective*	

4 Design Rationale and Relationships Among the Courses

The 1993-94 MSE Core Curriculum differs in several significant ways from earlier versions of the Core Curriculum and from most other software engineering curricula. Most existing programs are organized around the software lifecycle: they start with requirements elicitation and specification, and then progress through design, implementation, analysis and testing. In contrast, this curriculum is organized around topics that cut across the development process. It emphasizes the underlying principles and techniques (such as the use of formal models and the application of

good management principles) that can be applied in uniform ways to a broad spectrum of software development activities.

While the packaging of the material differs from most curricula, the overall content is similar, although with different emphases. Some topics that appear in several courses in other curricula (such as methods of software development) are covered in a single course in our curriculum. Others that appear in a single course in more traditional curricula (such as requirements elicitation and specification) are treated in several of the courses in the MSE curriculum. The rest of this section outlines the rationale for each course and some of the important relationships between them.

Models of Software Systems evolved out of a conviction that it is essential that software engineers have a coherent understanding of the fundamental mathematical models that underlie most of the good abstractions of software systems. Unlike courses in formal methods found in other programs, the course focuses less on specific notations than on the pervasive models on which those notations rest. Moreover, unlike many formal methods courses, it considers not only the use of mathematical models for software specification, but also certain models that underly testing, analysis, process modeling, and design selection. Thus the models course provides a "scientific" basis for the other courses in the program.

Methods of Software Development was designed to help students gain in-depth experience with techniques that span the gulf between a problem to be solved and a successful implementation. This course coalesces material often distributed across a number of distinct courses in other curricula: requirements specification, design, creation, maintenance. Among other things, the methods course builds on the notations and concepts introduced in the models course and demonstrates their practicality to real software systems development. We expect that among the methods treated in depth by the methods course, at least one will focus on the use of formal methods of software development. In addition, the methods course will be able to use the formal models to make distinctions between methods that are not themselves formal.

Management of Software Development focuses on organizational aspects of software development—project management, project planning, team organization, and process improvement. Its inclusion in the Core is based on the conviction that many of the problems in carrying out a successful software development effort can be traced to poor organization, poor planning, and poor understanding of the ways in which people can work together on a cooperative effort.

Analysis of Software Artifacts integrates various techniques for understanding the things produced by software engineers. It adopts a broad view of analysis, including topics often divided into separate courses, such as testing, formal verification, and techniques of static analysis. The analysis course builds strongly on the models course, by assuming that students have already learned the basic mathematical concepts that form the basis of many of the analytical techniques.

Architectures of Software Systems tackles head-on the problem of structuring large-scale software systems. It is concerned with system composition in terms of high-level components and interactions between them, rather than the data structures and algorithms that lie below module boundaries. While the field of software architecture is an emerging one, we believe that it will play an increasingly important role in software engineering.

The relationship between the methods and architecture courses is an in[illegible]ng one. While the methods course presents specific techniques and notations that span the complete developmental cycle of software, the architecture course focuses on the broader questions of software organization and high-level design. Typically, a specific method will traverse a narrow region of the architectural design space. (For example, object oriented methods use objects as the primary architectural building blocks.) By understanding the broader ("horizontal") architectural dimension students

can discriminate between different ("vertical") methods. Conversely, by understanding the context in which architectural design plays a role, students obtain a broader perspective on the ways good architectures can be put into practice.

Some topics that traditionally appear in separate courses are distributed across several in this curriculum. In particular, requirements elicitation and specification are handled by three aspects of the program. The management course covers requirements elicitation and in particular shows how the definition of a project's requirements impacts planning and cost estimation. The methods course introduces specific styles and techniques for specifying requirements. This is appropriate because the form of a requirements specification will generally depend on the methods chosen for software development. Finally, the development of a significant requirements document takes place in the context of the first semester Studio, which applies the concepts to a realistic problem.

5 Elective Courses and the Studio Project

Core courses occupy about 30% of the overall MSE curriculum. The rest is divided between elective courses and Studio.

- **Electives.** Given the huge amount of material that might be taught in a masters level software engineering program, any design of a Core Curriculum must necessarily exclude many topics. In the process of determining what to emphasize in the Core, the current curriculum design also considered areas that the Core does not cover in depth, but should be made available through electives. The most important of these are:
  - *Distributed systems*: The Core addresses some aspects of distributed systems development in (at least) the models, architecture, and analysis courses. However, students could benefit from a much deeper understanding of the design and implementation of distributed systems than the core courses can provide.
  - *Human-computer interfaces*: Design, implementation, and evaluation of user interfaces is a crucial part of many real software projects. While the core courses touch on some of the issues, we believe that most students could benefit from a course specifically targeted to this topic.
  - *Real-time systems*: Real time issues are introduced in the models and analysis courses. But neither of these courses aims to develop specific expertise in real-time systems design.
  - *Advanced management*: Many of the students in the MSE program will occupy management roles when they return to industry. While the management core course covers the basic material, there are many topics that could benefit from a more in-depth treatment. In particular, we would expect an advanced course to cover in-depth areas such as risk management and process evaluation.
- **The Studio.** An important consideration in the design of the MSE Core is the need to prepare students for the Software Development Studio. While Studio projects vary considerably in their details, generally they follow a similar high-level progression of activities: In the first semester students become familiar with the project domain and work with a customer to produce a statement of project requirements together with a project management plan that outlines how they plan to develop a system to meet the requirements. During the second semester students work through a design of their system, perhaps prototyping some aspects of it. During the summer they are primarily involved in implementation. During the

fourth semester they analyze and evaluate their experience, as well as maintain the delivered software.

The phasing of the core courses is designed to dovetail with this progression. The management course provides students with the skills to organize and plan their project. It also prepares them early in the semester to work with a customer to elicit requirements. Concurrently the methods course teaches students about alternative development methods so that they can make informed choices about the kind of software development that they will be using. An important part of the course is the specification of requirements; this allows students to have at hand appropriate notations and techniques for producing a requirements document for the Studio. By the second semester students will have learned to use several design methods and therefore be in a good position to carry out a design of their own. The software architecture course gives them added depth in the area of architectural design, and may actually use portions of a Studio project as an architectural design project. Concurrently, the analysis course helps students understand what kinds of static and dynamic analyses they can perform on their project deliverables.

6 Detailed Descriptions of the Courses

Models of Software Systems

Prerequisites:

Undergraduate discrete math including first-order logic, sets, functions, relations, proof techniques (such as induction).

Description:

This course covers formal models and logics for characterizing and reasoning about software systems. It considers many of the standard models for representing sequential and concurrent systems, such as state machines, algebras, and traces. It shows how different logics can be used to specify properties of software systems, such as functional correctness, deadlock freedom, and internal consistency. Concepts such as composition mechanisms, abstraction relations, invariants, nondeterminism, inductive and denotational descriptions are recurrent themes throughout the course.

This course provides the formal foundations for the other core courses. Notations are not emphasized, although some are introduced for concreteness. Examples are drawn from software applications.

Rationale:

Scientific foundations for software engineering depend on the use of precise, abstract models and logics for characterizing and reasoning about properties of software systems. Different kinds of mathematical models are suited to representing different kinds of systems. For example, state machines are often used to represent sequential systems; algebras, abstract data types; and traces, concurrent systems. Logics are used for describing and reasoning about behaviors of these models.

Objectives:

By the end of the course students will be able to:

- Understand the strengths and weaknesses of certain models and logics including state machines, algebraic and process models, and temporal logic.
- Select and describe appropriate abstract formal models for certain classes of systems.
- Describe abstraction relations between different levels of description, and reason about the correctness of refinements.
- Prove elementary properties about systems described by the models introduced in the course.

Viewpoint:

- Mathematical abstractions can be used to represent real systems.
- Formal reasoning can improve our ability to design and build systems by uncovering design flaws, validating implementations, precisely defining requirements.
- Different models have different strengths and weaknesses that determine contexts in which they are appropriately used.

Topic Outline:

1. Background:

   Sets, relations, sequences. Formal systems: syntax, semantics, proofs. Propositional and predicate logic.

2. State Machines:

   Finite state machines, Mealy/Moore machines, deterministic and non-deterministic machines, traces, invariants, relationship between state machines and programs, examples of use of state machines for reasoning about real systems.

3. Model-based Approaches:

   State spaces and state invariants, state transitions, pre- and post-conditions, introduction to Z notation and concepts, refinement and abstraction relations.

4. Algebraic models:

   Algebras, equational description, applications (particularly to specification of abstract datatypes), refinement.

5. Hybrid Models I:

   We consider the combination of algebraic models with use of pre- and post-conditions for defining operations (as exemplified by Larch), and perhaps the use of axiomatic specification in the context of a model-oriented approach (as exemplified by Z).

6. Model-oriented Approaches to Concurrency:

   Introduction to concurrency, limitations of state machines, Petri nets, reachability analysis, applications of Petri nets in areas such as project management, process modeling, and fault detection.

7. Trace models and Temporal Logics:

   Process algebras, CSP, CCS, use of non-determinism, trace specifications, linear and branching temporal logic, model checking.

8. Hybrid Models II:

   Combinations of trace models and algebras (as exemplified by Lotos).

Implementation Considerations:

This course should normally be taken in a student's first semester of the MSE.

Specific notations are introduced to illustrate the main ideas and give a concrete vehicle for description. However, the course does not seek to develop deep skills in using any specific notation or in integrating the use of the notation into a larger software development process. It is therefore critical that the (concurrently taught) course, Methods of Software Development, build on the ideas of this course to develop expertise in using one or more formal methods for software development.

Resource Requirements:

Many of the notations introduced in the course have associated tools that can help the students produce, check, and evaluate formal descriptions. (Examples of tools are the Fuzz type checker for the Z specification language, the FDR checker for CSP, and the Larch Prover for equational proofs.) The course expects to operate in an environment in which these tools (plus associated documentation, tutorials, and worked-out examples) are easily accessible.

Methods of Software Development

Prerequisites:

Models of Software Systems and Management of Software Development should be taken before or at the same time as this course.

Description:

This course will address techniques for bridging the gap between a problem statement and a software implementation. It focuses specifically on methods that guide the software engineer from requirements to code. The course will provide students with both a broad understanding of the space of current methods, as well as specific skills in using at least two of these methods.

Rationale:

In any engineering discipline the purpose of a "method" is to provide guidance in developing solutions to practical problems. Methods are useful because they structure the overall design problem through established notations, specific levels of abstraction, refinement techniques, documentation standards, analytical techniques, rules of thumb, and support tools.

A large number of software engineering methods are currently in use, such as object oriented methods, formal methods, Cleanroom development, prototype-oriented methods, and JSP/JSD. These methods differ in substantial ways including the notations they prescribe for describing requirements, designs, and implementations; their level of formality; the existence of associated development tools. Consequently methods have different strengths and weaknesses that determine when they are appropriate. To be an effective software engineer it is important to understand these strengths and weakness, to be able to assess when new methods might be appropriate, and, having selected a method, to be able to use it effectively.

Objectives:

By the end of the course, a student will be able to:

- Use at least two software engineering methods effectively.
- Make a critical assessment of the strengths and weaknesses of a broad range of methods.
- Understand the dimensions along which methods differ and understand the tradeoffs in making choices along those dimensions.

Viewpoint:

- Different methods have different strengths and weaknesses; no single method is good for all types of software development.
- Skill at using a particular method goes beyond knowing what notations are used in the method.
- Methods that provide strong analytical support will have an increasingly important role in software engineering.

Topic Outline:

The following topics will be interwoven throughout the course.

1. The Structure of Software Engineering Methods

   This topic is concerned with establishing a common foundation for understanding the similarities and differences between methods.

   - What is a method?
   - The design space for methods: notation, formality, coverage, phasing, potential for analysis, and tools
   - Criteria for evaluating methods

2. Specific Methods

   This part of the course develops at least two methods in depth. The specific methods will vary from semester to semester, but each of the methods will be considered in terms of the following issues:

   - Methodological ideas
   - Formal notions
   - Informal notions
   - Related methods
   - Associated tools
   - Exemplary case studies

3. Comparative Overview of Methods

   The course will provide a comparative analysis of existing methods, including well-established methods; currently fashionable methods; and those that have been proposed as improvements over current practice.

Implementation Considerations:

This course will normally be taken in a student's first semester.

The methods course builds on the notations and concepts introduced in the models course and demonstrates their practicality to real software systems development. Among the methods treated in depth by the methods course, at least one should focus on the use of formal methods of software development. In addition, the methods course can also use the formal models to make distinctions between methods that are not themselves formal.

Resource Requirements:

A significant portion of a student's time will be spent on applying specific methods to realistic problems in the context of lab projects. The course therefore requires that students have access to a solid development environment that supports (at least) the methods that are treated in depth. Additionally, it would be nice to have examples of the tools that are introduced for the other methods.

Management of Software Development

Prerequisites:

Students must have had industrial software engineering experience with a large project, or a comprehensive undergraduate course in software engineering.

Description:

This course provides the knowledge and skills necessary to lead a project team, understand the relationship of software development to overall product engineering, estimate time and costs, and understand the software process. Topics include life cycle models, requirements elicitation, configuration control, environments, and quality assurance, all of which are used broadly in other core courses and the Studio.

Rationale:

Professional software engineers are invariably called upon to participate in the business and management aspects of software development. Some of the standard management techniques apply. However, there are also many aspects of software development that make its management unique. Recently there has emerged a number of techniques for addressing these aspects. Systematically applied, these techniques can make the process of software production considerably more reliable and predicatable than it has been in the past.

Objectives:

At the end of this course the student will be able to:

- Write a software project management plan, addressing issues of risk analysis, schedule, costs, team organization, resources, and technical approach.
- Define the key process areas of the Capability Maturity Model, and the technology and practices associated with each.
- Define a variety of software development life cycle models, and explain the strengths, weaknesses, and applicability of each.
- Understand the relationship between software products and overall products (if embedded), or the role of the product in the organizational product line.
- Use software development standards for documentation and implementation.
- Understand the legal issues involved in liability, warranty, patentability, and copyright.
- Apply leadership principles.
- Perform requirements elicitation.
- Understand the purpose and limitations of software development standards, and be able to apply sensible tailoring where needed.

Viewpoint:

- Software development requires specialized management skills and techniques.
- Different development life cycle models are effective in different types of projects.
- A well planned and executed development process is essential to produce good quality software.
- Accurate requirements elicitation is the basis for all planning, size estimating, and costing.

Topic Outline:

1. Introduction

   This section characterizes the nature of software development, explains what makes it different from other types of product development, and places software in an overall product context. It also presents techniques for requirements elicitation and requirements management as a basis for forming estimates and plans.

   - The nature of software development management
   - Requirements elicitation
   - Requirements management and the statement of work
   - Software development process

2. Estimating Software Size and Cost

   This portion of the course considers the unique aspects of software development that affect productivity, how to elicit requirements and organize them in a way that supports the estimation process, and metrics for size and cost estimation. Consideration of the strengths and weaknesses of various metrics are a central part of this section.

   - Factors affecting software productivity
   - Software estimation metrics
   - Sizing software
   - Risks of current software estimation metrics
   - Adapting software estimation metrics

3. Planning and Scheduling

   This segment of the course considers how to write a practical software development management plan that includes realistic schedules, a resource usage plan, and supports effective team organization.

   - Software development standards and their limitations
   - Work breakdown structures
   - Making and using activity networks
   - Mechanized Gantt and calculating probability of completion
   - Resource allocation and training

- Software development team organizations
- Risk management
- Project management plans

4. Process Management

   This course segment concentrates on activities during the execution of the software development plan, including technical/management issues surrounding configuration management and quality assurance (such as tools, development techniques, and inspections). It also considers tracking the progress of the project and making adjustments as necessary. Interpersonal interaction and team management are also considered.

   - The role of configuration management
   - Implementing configuration management
   - The role of quality assurance
   - Implementing quality assurance (Cleanroom and inspections)
   - Causal analysis
   - Tracking, reviewing, adjusting goals, reporting; crashing a project
   - Professionalism and ethics
   - Leadership principles and self-management
   - Understanding subordinates and supervision
   - Managing sustaining engineering
   - Legal issues: copyright, patentability, liability and warranty
   - Product engineering

Implementation Considerations:

This course should be taken in the first semester of the MSE program.

The course is the primary carrier for techniques of requirements elicitation and some aspects of analysis. The Methods and Models courses are the primary carriers of requirements specification techniques. It will be important to synchronize the consideration of requirements specification in other courses with the completion of requirements elicitation in this course.

Resource Requirements:

The course involves several laboratory exercises, each requiring certain support materials and software. These are:

- Software size and cost estimation: Needs a requirements specification to serve as the basis for the assignment. Could benefit from a tool such as COCOMO.
- Activity network planning: Software (such as Microsoft Project) to assist in developing work breakdown structures, activity networks, and Gantt charts.
- Project planning: Word processing software with built-in graphics capability (such as Framemaker).

Analysis of Software Artifacts

Prerequisites:

Models of Software Systems plus experience programming a large software system.

Description:

This course will address all kinds of software artifacts—specifications, designs, code, etc.—and will cover both traditional analyses, such as verification and testing, as well as promising new approaches, such as model checking, abstract execution and new type systems. The focus will be the analysis of function (for finding errors in artifacts and to support maintenance and reverse engineering), but the course will also address other kinds of analysis (such as performance and security).

Various kinds of abstraction (such as program slicing) that can be applied to artifacts to obtain simpler views for analysis will play a pivotal role. Concern for realistic and economical application of analysis will also be evident in a bias towards analyses that can be applied incrementally. The course emphasizes the fundamental similarities between analyses (in their mechanism and power) to teach the students the limitations and scope of the analyses, rather than the distinctions that arose historically (static vs. dynamic, code vs. spec).

The course will balance theoretical discussions with lab exercises in which students will apply the ideas they are learning to real artifacts.

Rationale:

Our ability to build, maintain and reuse software systems relies on effective analysis of the products of software development. Analysis plays a role at all stages in software development and can be applied to all software artifacts, from requirements to code. Aside from its range of application, analysis is important because it usually makes few assumptions about how the artifact has been developed. In contrast to development methods, analysis techniques can be applied incrementally and without big initial investments or development constraints.

Objectives:

At the end of the course the student will:

- Know what kinds of analyses are available and how to use them.
- Understand their scope and power: when they can be applied and what conclusions can be drawn from their results.
- Have a grasp of fundamental notions sufficient to evaluate new kinds of analysis when they are developed.
- Have some experience selecting analyses for a real piece of software, applying them and interpreting the results.

Viewpoint:

- Testing and verification are not the only ways to analyze software artifacts.
- The analysis of specifications and designs is especially important.
- Abstraction may make analyses feasible that would otherwise not be cost-effective.
- The most important analyses are those that can be applied incrementally and can be extensively automated.

Topic Outline:

1. Overview of analyses:
    - The role of analysis in different phases of the lifecycle
    - Classification of analyses
    - Criteria: what makes a good analysis
2. Abstraction mechanisms
    - Automata abstractions (modes, hidden events)
    - Dataflow analysis and program slicing
    - Typing and data abstraction
3. Functional analysis
    - Random testing
    - Subdomain-based testing
    - Dynamic assertions
    - Verification
    - Simulation of specifications
    - Symbolic model checking
    - Abstract interpretation
    - Type systems
4. Performance analysis
    - Basic complexity theory
    - Queueing theory
    - Survey of resources
    - Profiling
5. Special topics
    - Security analysis
    - Reverse engineering
    - User interface analysis

Implementation Considerations:

Many of the topics covered in this course depend on the models introduced in Models of Software Development. This course should therefore follow the Models course.

Resource Requirements:

The course will require artifacts that can be used in course homework and projects for practicing the analysis techniques taught in class. In addition the course depends on having access to a number of analysis tools, such as: test coverage analyzers and mutation generators; a theorem prover; a model checker; automata simulators (e.g., Statemate); a program slicer; a profiler; a debugger; etc.

Architectures of Software Systems

Prerequisites:

Experience with at least one large software system. Could be satisfied by industrial software development experience or an undergraduate course in software engineering, compilers, or operating systems.

Description:

This course introduces architectural design of complex software systems. The course considers commonly-used software system structures, techniques for designing and implementing these structures, models and formal notations for characterizing and reasoning about architectures, tools for generating specific instances of an architecture, and case studies of actual system architectures. It teaches the skills and background students need to evaluate the architectures of existing systems and to design new systems in principled ways using well-founded architectural paradigms.

Rationale:

As the size and complexity of software systems increases, the design problem goes beyond the algorithms and data structures of the computation: designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; composition of design elements; scaling and performance; and selection among design alternatives. This is the *software architecture* level of design.

In other courses students are exposed to one or two specific application architectures (such as for a compiler or for parts of an operating system), and they may hear about a few other architectural paradigms. But no serious attempt is made to develop comprehensive skills for understanding existing architectures and developing new ones. This results in a serious gap: students are expected to learn how to design complex systems without the requisite intellectual tools for doing so effectively. This course helps bridge this gap by bringing together the emerging models for software architectures and the best of current practice.

Objectives:

This course teaches students how to design, understand, and evaluate systems at an architectural level of abstraction. By the end of the course a student will be able to:

- Recognize major architectural styles in existing software systems.
- Describe an architecture accurately.
- Generate architectural alternatives for a problem and choose among them.
- Construct a medium-sized software system that satisfies an architectural specification.
- Use existing definitions and development tools to expedite such tasks.
- Understand the formal definition of a number of architectures and be able to reason precisely about the properties of those architectures.
- Use domain knowledge to specialize an architecture for a particular family of applications.

Viewpoint:

- Architectural design is a different kind of activity from programming.
- Many systems can be understood as examples of common architectural styles of system organization.
- The same system can be designed using many different architectures: however, the choice will have a significant impact on the properties of the resulting system.
- Software architectures can be described formally.

Topic Outline:

1. What is Software Architecture?
   - *Overview.* The architectural level of software design.
   - *What is a Software Architecture?* Basic elements of software architecture, together with an overview of commonly-used kinds of architectures.
2. Classical Organizations & Module Interconnection Languages.

   Overview of traditional approaches to software system organization, emphasizing languages for program modularization.
3. Procedure Call Organizations
   - *Subroutines and Objects.* Main program/subroutine organization and approaches to information hiding, abstract data types, and objects.
   - *Formal Models.* An introduction to formal models of software architecture.
4. Data Flow Organizations
   - *Batch Sequential Systems.* System organizations based on sequential processing of information.
   - *Pipes & Filters.* Pipe/filter organizations consisting of data transformers (filters) connected by data streams (pipes).
   - *Data Flow Implementations.* Implementation techniques for Unix-style pipes and filters.
5. Event-based Organizations
   - *Event models.* System organizations based on event broadcast allow loose coupling between components while still supporting integrated processing.
   - *Implementation of Event Systems.*
6. Threads and Processes
   - *System Based on Multiple Processes.* Techniques of system organization based on interprocess communication and multiple (concurrent) threads of control.
   - *Formal Models of Concurrent Processes.* Formal models for reasoning about processes.
7. Repository-Oriented Systems

- *Database Architectures.* Systems organized around a shared data repository. Transactions.
- *Blackboards and Others.* Blackboard and other specialized repository architectures.

8. Case Studies: Several case studies of real systems such as:
   - *Instrumentation Software.*
   - *Information Systems.*
9. Connection
   - *Beyond Module Connection Languages.* Advanced notations for modules: adding semantics to interfaces, externalizing connectors, and coordination languages.
   - *Interface Matching.* Solutions to problems of component signature mismatches.
   - *Mediators and Wrappers.* Solutions to mismatches when the discrepancy is deeper than a signature.
10. Design Assistance
    - *Rule-based Assistance.* A technique for making architectural design explicitly dependent on the functional requirements.
    - *Design Selection.* Other techniques for automated design selection.
    - *System Generators.* Automated support for system generation: environment generation, application generation, and architectural design environments.
11. Domain-Specific Architectures:

    Example architectural styles that exploit the constraints of a specific application domain such as: layered communication protocols, architectures for mobile robotics, human-computer interfaces, and computer-based medical systems.

Implementation Considerations:

This course should normally be taken in the second semester of the MSE program.

Resource Requirements:

Students in this course carry out a number of system building exercises. These exercises can exploit at least the following kinds of software development support: module-oriented programming languages and environments; component libraries; event-based integration systems. Additionally, it is helpful to have support tools for the formal notations introduced in the course. Well-documented examples of real systems for architectural analysis would be quite useful.