The CMU Master of Software Engineering Core Curriculum*

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

†School of Computer Science and ‡Software Engineering Institute
Carnegie Mellon University
Pittsburgh PA 15213

Abstract. This paper outlines the new Core Curriculum of the Carnegie Mellon University Master of Software Engineering Program. Unlike most MSE curricula, which typically organize their courses around aspects of a software development lifecycle, this curriculum focuses on the cross-cutting disciplines of modelling, problem solving, management, analysis, and design.

1 Introduction

Most professional masters in software engineering (MSE) programs are organized, in part, around a set of core, required courses. These courses are typically designed to cover the "basics" of software engineering, provide a common foundation for more advanced courses, and develop certain essential skills that every software engineer should know.

A key issue in organizing such a core curriculum is what should go in it. This raises questions of content, depth, focus, and organization. What is the essential technical basis for software engineering? What is an appropriate balance between depth and breadth of coverage? What kinds of practical skills, if any, should students learn? How should the material be distributed across the courses, and in what order should it be taught?

In most programs the answers to these questions are embodied in a curriculum organized around phases of a software development lifecycle. For example, there will typically be a course in requirements analysis, a course in specification, one in design and implementation, one in testing, and so on.

This organization has a number of advantages. First, there is a clear rationale behind it: each of the topics has its place in software development practice. Second, it

* This research was sponsored by the National Science Foundation under Grant Number CCR-9357792, by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330, and by Siemens Corporate Research. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Wright Laboratory, the U.S. Government, or Siemens Corporation. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon.

is a well-established path: software engineering textbooks are often organized in this way, and there are dozens of textbooks on each of the phases. Third, each course can operate largely in a stand-alone fashion. This makes the curriculum ideal for part-time students.

On the other hand it also has a number of disadvantages. First, it tends to emphasize a waterfall model of software development. Second, it fails to highlight common techniques and principles that underly many aspects of software development, such as abstraction techniques, analysis, modelling, etc. Third, it tends to compartmentalize each topic. For example, students who learn a state machine notation for specification may fail to realize the same ideas can be applied to algorithm development, program testing, fault tree analysis, and many other aspects of software development.

In this paper we outline a different approach, one that focuses on the cross-cutting principles of software development. Instead of concentrating on development stages, this curriculum is oriented around topics that pervade all aspects of software development. Specifically, the core curriculum is organized around the topics of modelling, methods of development, management, analysis, and architecture. In the following sections we describe the intended content of each of these five core courses and explain their rationale. Next we show how these courses sit within the context of a broader MSE curriculum. Then we provide a more detailed account of each core course, highlighting its content, prerequisites and resource requirements. Finally, we describe our experience of teaching this core in the Carnegie Mellon MSE Program during the 1993-94 school year, and use this experience to evaluate the effectiveness of the approach.

2 The CMU Master of Software Engineering Program

The Carnegie Mellon University MSE Program was founded in 1989 as a joint program between the School of Computer Science and the Software Engineering Institute. It is a four-semester, intensive program for professional software engineers, and leads to a terminal masters degree. The goal of the program is to develop technical leaders in industrial software engineering. These people should be able to act as agents of change in their respective organizations, and be able to apply to software development both the best of current practice as well as emerging technologies. Consequently, students who are accepted to the program must have both a strong background in computer science as well as two years' industrial software development experience that indicates strong potential for leadership.

The incoming class size for the program is usually about 20 students. In the past these students have had an average of about 5 years of industrial experience. About half of the students come from large corporations (Digital Equipment, HP, Westinghouse, General Motors, etc.), while the others represent a variety of smaller software development firms. Many of the students are supported by their employers, who expect them to return to the company after finishing their degree.

It is worth noting that both the student body and the broad program mission differ from that found in a typical program. Most software engineering programs are organized to meet the growing demand for software engineering practitioners.

They primarily serve a local population of professionals interested in part-time study. These students expect to apply what they learn immediately. In contrast, CMU's software engineering students come from all over the world and are full-time resident. Although the students in the CMU MSE do expect to learn the best of current practice, additionally the program is aimed at giving them the technical basis for dealing with the rapidly evolving discipline of software engineering, and for helping put into practice new techniques as they become available. This demands a curriculum strong in both fundamental computer science and beyond-leading-edge material in software engineering.

The MSE program is organized around three basic components: the Core Curriculum, a number of elective tracks, and the Software Development Studio. The 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. 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. In that respect, the Studio is similar to the design studios that characterize architectural degree programs.

3 Evolution of the Core Curriculum

The CMU MSE was initially organized around the curriculum recommendations developed at the Software Engineering Institute (SEI). The SEI arrived at its recommendations through a process that reflected software development itself. In February, 1986, a workshop was held to glean ideas about the content of software engineering from leading educators and practitioners. The results of this "requirements elicitation" activity was a list of topic areas to be included in any curriculum [FGT87]. These topics were arranged in six core courses by an invited group of experienced software educators. First published in 1989, the SEI recommended a curriculum to serve as a basis for establishing programs rooted in the current best practice. It is a combination of these six core courses, a project course, and electives [AF89].

CMU's Department of Computer Science began to study the possibility of initiating an MSE program in 1986. From January to May of 1989, a joint SEI/SCS planning committee adapted the six SEI core courses into five, and added a course in advanced topics in computer science to strengthen the technical basis of the program. It also did detailed planning of what came to be called the Software Development Studio [AGHT89, Tom91]. The program was taught using this curriculum from 1989-1993.

While there were many positive features of the program as a whole, there was a general sense that the initial core curriculum could be substantially improved. In particular, we felt that we could do a better job at addressing in a coherent way some of the common techniques and concepts that were distributed throughout the

curriculum. In the spring of 1993, the authors were charged with developing a new core curriculum for the program. In the remainder of this paper we describe this design.

4 Synopsis of the Core Curriculum

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

1. **Models of Software Systems:** This course treats foundations for software engineering based on the use of precise, abstract models and logics for characterizing and reasoning about properties of software systems. Specific notations are not emphasized, although some are introduced for concreteness. The main topics include state machines, algebraic models, process algebras, trace models, compositional mechanisms, abstraction relations, temporal logic. Illustrative examples are drawn from software applications.
2. **Methods of Software Development:** This course addresses the practical development of software using methods that help bridge the gap between a problem to be solved and a working software system. The intent of the course is to introduce students to comprehensive approaches to Requirements Analysis, Design, Creation, and Maintenance. Representative methods and notations include: object-oriented methods, JSD/JSP, VDM, Z, Larch, Structured Analysis/Design, Cleanroom development, and prototype-oriented development. In particular, students gain in-depth experience with three specific design methods, and are expected to understand the scope of applicability of each method. The course also introduces students to the use of supporting tools.
3. **Management of Software Development:** This course focuses on the management and organization of resources – both human and computational – for large-scale, long-lived software development projects. It treats the management of individual software development efforts and long-term capability improvement, including 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:** This course focuses on the analysis of software development products including delivered code, specifications, designs, documentation, prototypes, test suites. It treats both static and dynamic analyses, such as type checking, verification, testing, performance analysis, hazard analysis, reverse engineering, and program slicing. Tools for analysis are used where appropriate.
5. **Architectures of Software Systems:** This course is concerned with the design of complex software systems at an architectural level of abstraction. It treats organization of complex software based on system structure and assignment of functionality to design components. The main topics include common patterns of architectural design, tradeoff analysis at an architectural level, domain-specific architectures, automated support for architectural design, and formal models of software architecture.

5 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*	

6 Design Rationale and Relationships Among the Courses

As noted earlier the MSE Core Curriculum differs in 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 more precise vague notions embodied in informal methods.

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, model checking, formal verification, and some techniques of static analysis (such as slicing). 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 interesting 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.

7 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.

8 Detailed Descriptions of the Courses

8.1 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.

8.2 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.

8.3 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).

8.4 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.

8.5 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.

9 Experiences Teaching the Core

Having now taught the Core Curriculum once, we are in a position to provide a preliminary evaluation of its effectiveness. Overall, we believe that the new curriculum is an improvement over the previous design. We are pleased with the basic organization of topics and have found that the current design provides flexibility for achieving a balance between concrete, immediately-applicable skills and longer-term appreciation of emerging technologies. On the other hand there are a number of areas that we think could use some improvement. These are detailed below.

Students comment that they appreciate the material taught in the Core. They also enjoy the point of view from which it is taught. Their two areas of concern are

the workload and the connection between the Core content and current practice. Balancing load across the courses can be difficult: most of the core courses involve projects and there are inevitable overlaps in peak demand from these assignments. In terms of applying the core material to current practice, it can be difficult to find the right balance between innovative technical material that may not yet be well-integrated into software engineering practices, and concrete skills that can be applied today. To improve the situation we are attempting to improve integration of the Core with the Studio activities—drawing on the Studio for assignments and working with students in the Studio to help them apply the ideas they have learned in the Core.

Models of Software Systems achieved its goal of providing a basic formal background for describing and reasoning about software systems. One measure of its success has been the extent to which other courses could leverage the material that it introduced. For example, Methods of Software Development was able to build on the introduction to Z and on CSP. Analysis of Software Artifacts built on the concepts of state machines, Z, temporal logic and, traces. Architectures of Software Systems used both Z and CSP to model software architectures. Furthermore, students have been able to adapt some of the modelling techniques for use within the studio [Gar94].

The main problem with our initial offering of Models was the proliferation of notations. The intent of the instructors was to emphasize the underlying mathematical models. However, to do this requires the introduction of concrete notations. The problem was that each kind of model required a different language. Consequently students spent too much time learning notations and the associated techniques for using them effectively. We believe that we can do better by starting with a generic formal basis and using this as a foundation to show how specific formal models can treated as specializations.

Methods of Software Development was largely successful in exposing students to several techniques for obtaining software solutions. In the initial offering we introduced students to an object-oriented method (Objectory), a formal method (Z), and JSP/JSD. Each of these was organized around a common case study in which students developed a system design using the specific method.

There were two main problems that we encountered in teaching this course. The first was the difficulty in covering three different methods at the level of depth that we desired. Since a major goal was for students to become proficient at each of the methods in the course, each method required a large investment of time to master the notation, learn to use development tools, work out a number of detailed examples, etc. It was difficult to pack all of this into one semester. The obvious alternative is to cover fewer methods.

A second problem was a dissatisfaction with the methods themselves. Most methods advertise themselves as complete packages, capable of handling all aspects of software development. In reality, of course, specific methods are good for certain classes of system, and for certain aspects of their development. Thus it is important to be able to combine methods, perform cost/benefit analyses for method selection, and choose aspects of methods that work best for a given problem. This is an unsolved problem, but we hope at least, if not to solve it, to articulate it more clearly in future offerings of the course.

Architectures of Software Systems has evolved gradually over the last three years. We are pleased with its current form and content. (Details of the course can be found elsewhere [GSO+92, GS94].) Students report that the course changes the way they look at systems, and generally feel that the course prepares them well for the system design component of the studio.

Our only significant difficulty in teaching the course has been in finding good materials for use in assignments. The course includes a number of development tasks that require students to use different architectural styles to implement a system. It has been difficult to find tasks that can be done in two or three weeks, while at the same time providing interesting architectural design challenges. A related issue is the general lack of well-documented case studies of systems that can be used by students for understanding how others develop systems.

Analysis of Software Artifacts succeeded in its basic of aim of conveying techniques that could be applied to real problems. The students were given four small projects, all based on the analysis of a pre-existing artifact that had not been contrived for the course. They found bugs in an Ada implementation of a traffic light using the SMV model checker; they uncovered, with formal verification, a number of dangerous subtleties in two string handling routines of the C library; and they found all the seeded bugs (and one more!) in the code of the UNIX grep utility using the GCT test coverage tool. Model checking was by far the most popular technique. In a final project to investigate a further technique, or apply one they already learned to a new problem, a third of the class chose to pursue model checking.

One project failed miserably. The students were asked to write a small formal specification of the UNIX directory graph with some basic operations, and prove some simple properties. Despite valiant efforts, most were unable to produce a specification that was simple enough to admit any kind of formal proof. The verification problem caused much anguish too, and several students produced lengthy proofs that overlooked fundamental issues. Both projects suffered from the same problems: ideas (such as precondition calculation, loop invariants, modelling aliasing, etc) that are probably too subtle to master in a couple of weeks, and a mismatch between what is needed to solve real problems and the state of textbook examples. Even the instructor took almost a day to complete a proof of correctness of the string concatenation routine.

Finding good material on testing was a big problem. There were no suitable textbooks, so we relied on a selection of academic papers. Unfortunately, although these provide considerable insight (in, for example, the relative merits of random and subdomain testing or the ranking of coverage criteria), their results are almost entirely negative. Even when they provide positive results (such as how to obtain good subdomains), applying these results is far from straightforward, it rarely being clear how to obtain any immediate practical benefit.

Management of Software Development required the least effort to implement because of its close relationship to both the SEI curriculum and its predecessor in the MSE program. In fact, the course is taught via the National Technological University distance learning network as an SEI-sponsored course at the same time the MSE students participate in it. The most difficult part of the revised course was the inclusion of requirements elicitation techniques. Since the requirements of a project are used as input to the project planning process, it is logical to discuss requirements

acquisition and requirements management in this course. However, the syllabus is crowded, and there is not enough time for students to gain practical experience with elicitation techniques. In the future we plan to use the Studio more fully for giving students hands-on experience with requirements elicitation, specification, and analysis.

Another problem area in software management in general is the use of metrics for predicting project cost and developing a project schedule. Students tend to treat all data as sacred, regardless of the source. They also do not seem to have sufficient background in statistics to properly interpret even "good" data. We are thinking about requiring some probability and statistics as prerequisite material, or finding ways to incorporate it into the curriculum.

10 Conclusion

The new Core Curriculum of the CMU MSE can be viewed as an experiment in ways to teach software engineering more effectively. Based on our limited experience thus far, we are pleased with the results and enthusiastic about its long-term potential. While there are many areas that need improvement, overall we believe that the approach we have adopted is worth serious consideration by others.

Acknowledgements

We would like to thank the many students and faculty who have contributed to the MSE program. Their comments and constructive criticism were key ingredients in the development of the core curriculum.

References

[AF89] Mark A. Ardis and Gary Ford. 1989 SEI report on graduate software engineering education. Technical Report CMU/SEI-89-TR-21, Carnegie Mellon University Software Engineering Institute, 1989.

[AGHT89] Mark A. Ardis, Norman E. Gibbs, A. Nico Habermann, and James E. Tomayko. The Carnegie Mellon University Master of Software Engineering Degree Program. Internal document., 1989.

[FGT87] Gary Ford, Norman E. Gibbs, and James E. Tomayko. Software engineering education: An interim report from the Software Engineering Institute. Technical Report CMU/SEI-87-TR-8, Carnegie Mellon University Software Engineering Institute, 1987.

[Gar94] David Garlan. Integrating formal methods into a professional master of software engineering program. In *Proceedings of the Z Users Meeting*, Workshops in Computing. Springer-Verlag, June 1994.

[GS94] David Garlan and Mary Shaw. Software development assignments for a software architecture course. In *Software Engineering Resources: Proceedings of the ACM/IEEE International Workshop on Software Engineering Education*, May 1994. Imperial College DoC Technical Report 94/6.

[GSO+92] David Garlan, Mary Shaw, Chris Okasaki, Curtis Scott, and Roy Swonger. Experience with a course on architectures for software systems. In *Proceedings of the Sixth SEI Conference on Software Engineering Education*. Springer Verlag, LNCS 376, October 1992. Also available as CMU/SEI technical report, CMU/SEI-92-TR-17.

[Tom91] James E. Tomayko. Teaching software development in a studio environment. *SIGSCE Bulletin*, 23(1):300–303, March 1991. Papers of the 22nd SIGSCE Technical Symposium.