The Software Factory: An Undergraduate Computer Science Curriculum

John D. Tvedt¹, Roseanne Tesoriero², and Kevin A. Gary³
¹Department of EE & Computer Science, The Catholic University of America (CUA), Washington, DC, USA, ²CUA and Fraunhofer Center for Experimental Software Engineering, College Park, MD, USA, and ³UNICON, Inc., Chandler, AZ, USA

ABSTRACT

Industry often complains that current academic curricula fail to address the practical issues of real software development. This paper outlines a proposal for an innovative core curriculum for a Bachelor of Science in Computer Science. The proposed core curriculum contains elements of traditional computer science programs combined with software engineering via a team-oriented, hands-on approach to large-scale software development. In addition to traditional lecture/project/exam courses, students are required to take an eight-semester sequence of ‘Software Factory’ courses. Software Factory courses put the students’ newly acquired skills to work in a real software organization staffed and managed by all students in the program. Students from all courses in the Software Factory sequence meet simultaneously to fulfill their roles in the software organization. We expect that the students will be better prepared software engineering practitioners after completing a curriculum that combines traditional courses with practical Software Factory experience.

INTRODUCTION

With the explosive growth of the Internet and the permeation of software into nearly every aspect of our lives, the need for qualified software developers to build quality software is apparent. Industry would like educational institutions to train future employees in the latest technology. However, the value system of academic institutions emphasizes long-term education rather than training in short-term skills. While it may be tempting to focus on the latest technologies to satisfy industry demand, from a pedagogical perspective,
doing so would not serve students in the long term. Once the latest technology becomes obsolete, so does the students’ knowledge. Industry complains that current academic curricula fail to address the practical issues of real software development (Coulter & Dammann, 1994; Dawson, Newsham & Kerridge, 1992; Gibbs, 1989; Shaw, 1990; Wasserman, 1996). With current, undergraduate, computer science curricula, it is rare for students to encounter large-scale development requiring teamwork, written and oral communication skills, maintenance, management, and quality activities. Additionally, traditional computer science courses do not expose students to the non-technical issues that often drive decision making in a real development environment.

Academic institutions have attempted to address industry complaints with curricula in computer science. Assigning team projects in semester courses (Dawson & Newsham, 1997; Easterbrook & Arvanitis, 1996; Hilburn, 1997; Horning & Wortman, 1997), team projects that span multiple semesters (Garlan, Glutch, & Tomayko, 1997; Moore & Potts, 1994), and encouraging internships (Powell, Diaz-Herrera, & Turner, 1997) are a few of the ways academic institutions have attempted to include practical experience along with traditional coursework. However, project courses and internships often begin late in a student’s career and are typically short in duration. The exposure does not provide the depth of experience to appreciate the responsibilities of the roles and the implications of their decisions on future development. Nor, do the students have the opportunity to learn from their mistakes and apply their experience to future projects.

The Catholic University of America (CUA) is a small, private institution in the heart of Washington, DC, which has one of the nation’s fastest growing technology sectors. This growth is fueled by an established government services market and a rapidly expanding telecommunications market. Of course, much of the technology developed for these sectors has as its core, complex software. Like many higher educational institutions these days, CUA has difficulty in retaining and motivating students who see better immediate opportunities in these markets. The challenge facing CUA today is one echoing through the halls of Computer Science departments throughout the country (if not the world), “How can we make our curriculum meaningful in today’s technology-driven world without compromising the essential knowledge and training a student in computer science must receive?”

The purpose of this paper is to describe an answer to this challenge, the Software Factory. The Software Factory combines traditional computer science coursework with experience-based learning in which students participate in the
development of several, real-world software projects. Each student gains experience in every participatory role in the software life cycle, from serving as a requirements analyst to a software tester to a project manager, and all roles in between. In this paper, we present the Software Factory curriculum.

The main objectives of the Software Factory curriculum are the following:

1. meet industry needs for producing computer scientists familiar with today’s technology and processes;
2. ensure computer science students are given a solid and lasting foundation in computer science by providing an accredited computer science program;
3. attract and retain high quality students;
4. conduct empirical software engineering research; and
5. encourage multidisciplinary collaboration.

THE SOFTWARE FACTORY

The Software Factory concept combines traditional computer science coursework with Software Factory courses. The traditional courses cover the fundamental topics of computer science (see Appendix A). Software Factory courses expose students to large-scale, team-oriented development in a software development organization staffed and managed by students under the guidance of faculty. There are eight Software Factory courses that combine to form a software development organization. Each course represents a specific, software engineering role or job within the development organization. The eight-semester sequence progresses the students through the following roles: (1 & 2) Software Factory process and tools trainee, and software system tester, (3 & 4) software developer and maintainer, (5 & 6) requirements analyst, test planner and software designer, and (7 & 8) software project manager. Students from all courses in the Software Factory sequence meet simultaneously to fulfill their roles in the software organization. The enrollment in the program allows for multiple teams within the organization.

Software Factory courses are hands-on courses that require student participation in the Software Factory throughout their undergraduate career. The Software Factory is a software organization staffed and managed by students in the Computer Science program. Software Factory courses are facilitated by an instructor, but they emphasize learning through real work experience. These classes meet twice each week. One class meeting lasts 1 hr...
and is led by the instructor/‘consultant’. During this meeting, the ‘consultant’ introduces concepts that are relevant to the current work being performed in the factory and addresses problems faced by the students. The second class meeting lasts 2 hr. During this second meeting, all Software Factory classes meet simultaneously in one location, thus fully staffing the factory. During this session, the instructor is a ‘facilitator’ who does not decide right or wrong (as in traditional courses), but instead facilitates learning the pitfalls and peaks in development processes. The facilitator may perform various roles external to the organization, such as ‘customer,’ ‘patent agent,’ ‘end-user,’ ‘certification agent’ (health-care, aviation, etc.), and so forth.

The purpose of the Software Factory is to provide students with practical experience in software development. The students should gain business experience, as well as technical experience. It is important for the students to be exposed to the constraints that business decisions place on technological decisions. The classroom space for the Software Factory simulates a real working environment with cubicles, meeting rooms, and office equipment.

The projects for the Software Factory will be chosen by the managers (fourth-year computer science students). These projects may reflect current trends in industry. For example, the Internet has created unique opportunities for students in the Software Factory to gain experience in both new technologies and entrepreneurial areas. The students could create e-commerce websites to provide services and software. One potential project for the Software Factory would be to build and run an online auction site. The site could be targeted at students, allowing them to auction used textbooks, or other small items. Revenue from the web site could be used to support the Software Factory.

Additional projects might be related to ongoing research at the academic institution. The students could negotiate with the faculty to develop software that would support their research projects, thus, encouraging multidisciplinary collaboration. The Software Factory could negotiate with industry for projects that would promote academic/industry partnerships.

**Software Factory Course Sequence/Description**
Each course in the eight-semester sequence through the Software Factory is described below.

*1st Semester*
CSC 151: Software Development Process, Tools, and Testing I (3)
This course introduces students to the software development process and the tools that support it. Students learn about software processes, in general, as well as the process in use at the factory. These students learn about the roles and activities of the members of the factory, such as developers, testers, quality assurance, and management. The students learn about, and use, the tools that support the roles and activities at each different stage in the software development process.

In addition, students are put in the role of software testers. Student responsibilities include: writing test plans that test the requirements, writing test cases, running test cases, documenting test results, and following the documented process and standards.

2nd Semester
CSC 152: Software Development Process, Tools, and Testing II (3)

This course is a continuation of CSC 151.

3rd Semester
CSC 251: Software Development and Maintenance I (Code and Unit Test) (3)

In this course, students are put in the role of software developers. Student responsibilities include: writing software that adheres to the design, unit testing, integration testing, documenting, performing peer reviews, and following the documented process and standards.

4th Semester
CSC 252: Software Development and Maintenance II (Code and Unit Test) (3)

This course is a continuation of CSC 251.

5th Semester
CSC 351: Software Requirements, Test Planning, and Design I (3)

In this course, students are put in the role of software requirements analysts and test planners. Student responsibilities include: meeting with customers, analyzing customer requirements, writing a requirements specification document, writing test plans, performing peer reviews, and following the documented process and standards.

In addition, students are put in the role of a software designer. Student responsibilities include: writing a design document that meets the require-
ments specification, performing peer reviews, and following the documented process and standards.

6th Semester
CSC 352: Software Requirements, Test Planning, and Design II (3)
This course is a continuation of CSC 351.

7th Semester
CSC 451: Software Project Management I (3)
In this course, students are put in the role of a software project manager. Student responsibilities include: project planning, resource allocation, project estimation, project tracking, risk analysis/mitigation, personnel management, SQA, and planning the future direction of the Software Factory.

8th Semester
CSC 452: Software Project Management II (3)
This course is a continuation of CSC 451.

DISCUSSION

We believe that the Software Factory concept meets the objectives listed in the INTRODUCTION. This section describes how the design of the Software Factory meets each objective.

Meet the Needs of Industry
The needs of industry addressed by the Software Factory design include: exposing students to new technologies, teamwork, large-scale development, management activities, maintenance activities, quality activities, and written and oral communication. By having the students meet in one location simultaneously, we simulate a real world, development organization. By participating in the different roles of the Software Factory, students are learning the skills needed by industry as well as gaining an appreciation for their use in industry. For example, configuration management can be taught in a traditional course, but when a student is confronted with a large-scale system, the need for configuration management is better appreciated. The structure of the Software Factory allows the courses to be flexible and adaptive to new technology.
Create an Accredited Program
The coursework designed for the Software Factory follows guidelines for software engineering education (Bagert, 1998; Bagert et al., 1999; IEEE/ACM Software Engineering Coordinating Committee, 1998) and meets the ABET computer science curriculum requirements (2000). A mapping from the requirements to the coursework is given in Appendix B.

Attract and Retain Quality Students
Students often complain that they do not get enough exposure to coursework in their major until later in their academic careers. The Software Factory concept immediately immerses students into their area of academic interest. By offering a unique program that gives students an opportunity to have early exposure in their area of concentration, we can attract quality students. With exposure to the latest technologies and interesting projects and opportunities, we hope to be able to retain those students.

Conduct Empirical Software Engineering Research
The Software Factory will benefit the computer science faculty as well. With a fully staffed Software Factory, there will be multiple teams working within the organization. Those teams could work independently to develop distinct versions of the same product. This setup would provide an opportunity to conduct empirical software engineering studies. For example, an experiment could be run to test the benefits of implementing software inspections. Some teams in the Software Factory could act as a control group while others could develop the same product utilizing inspections.

Encourage Multidisciplinary Collaboration
Faculty are confronted often with the need for software developers when working on and applying for research grants. As with industry, often the supply does not meet the demand. The Software Factory could provide resources to support those grants. The students would be exposed to problems from various disciplines. The end result is an environment that encourages collaboration among the faculty of the academic institution.

IMPLEMENTATION PLAN
Implementing the Software Factory concept requires special consideration. The physical environment, instructional methods, and grading techniques vary
significantly from traditional courses. This section gives guidance on possible implementation strategies for the Software Factory curriculum.

**Physical Environment**
Each team will have its own laboratory session in which the students on the team will staff the Software Factory. The physical location for the Software Factory laboratory must be able to accommodate 20 people. The laboratory should be set up like an office space with meeting rooms, cubicles equipped with computers and miscellaneous office equipments and supplies.

**Flow of Work**
We have considered two life cycle process models for use in the Software Factory: waterfall and evolutionary. Using the waterfall process model each project will have a 2-year cycle time (Fig. 1).

The managers will decide which projects the Software Factory will accept during the first few weeks of the first semester. Upon acceptance, those projects will undergo requirements analysis, test planning, and design by the third-year students during the academic year. The managers will be responsible for guiding and measuring the development and testing of the projects for which they wrote the requirements and design in the previous year. The development and maintenance will be performed by the second-year students. The testing will be performed by the first-year students (second semester). The trainees (first semester first-year students) will be assigned to one of the developers as a mentor.

![Fig. 1. Waterfall project work flow.](image-url)
to learn about the process and the tools used in the Software Factory. The trainees will be exposed to all of the different software engineering roles.

We also have an option to implement an evolutionary life cycle process model. We decided to create this option for two reasons. The first reason is that some projects simply do not lend themselves to waterfall development, they are more evolutionary in nature. The second reason is that due to the long cycle time of the waterfall model proposed, it will be more difficult for the students to see the impact of their decisions. For example, the impact of poor requirements may not be felt for more than a year using the waterfall model and, thus, the students will not learn from this mistake for more than a year. Using the evolutionary process model each project will have increments with cycle time on the order of weeks, not years (Fig. 2).

The student roles remain essentially the same as for the waterfall model. The managers still decide which projects the Software Factory will accept during the first few weeks of the first semester. Upon acceptance, those projects will undergo requirements analysis, test planning, and design by the third-year students during the academic year. Rather than creating a single version of these documents during the academic year, though, these documents will evolve over a number of increments. Short increments will allow immediate feedback on ambiguous requirements, test plans, and design documents from developers and
testers. The managers will still be responsible for guiding and measuring the development and testing of the projects, but the requirements and design for the project will be created immediately prior to these activities, not in the previous year. The short increments will allow the managers to compare measurements from multiple increments, and perhaps improve the process and project planning after analyzing these data. The development and maintenance will be performed by the second-year students. Short increments will allow the developers to get immediate feedback on the quality of their work. The testing will be performed by the first-year students during the entire year, not just in the second semester. Their training will be the same as for the waterfall model, but it too will last throughout the academic year.

**Faculty Role**
The faculty have a role in the Software Factory that is quite different from traditional lecture courses. The faculty must act as a facilitator and as a consultant. The faculty must guide the students in the use and improvement of the development process, while consulting with them on all aspects of the project they are developing. Each Software Factory course will not have a well-defined syllabus, but rather a well-defined process for learning (Fig. 3).

There are four phases in the learning process: Understand Process, Apply Process, Identify Process Weaknesses, and Modify Process. At any point in time during the course, students will be in one of the learning phases. The faculty will facilitate discussion in class helping students in each phase of the continual learning process.

Fig. 3. Continual learning process.
All students will start the academic year in the Understand Process phase. Each role within the Software Factory will have a process that students must follow. During this phase the faculty are responsible for providing students with existing process documents. After the students have read the process documents the faculty will guide discussions in class in order to clarify any misconceptions that the students may have concerning what they have read. A large portion of these discussions will consist of individual students teaching the rest of the class about the process.

After the students understand the process, they will apply it to a particular problem in the Apply Process phase. For example, developers will start to develop code once they understand the process for software development. During this phase the faculty are responsible for consulting with students on the application of the process. This consultation will occur during class time in the form of discussions on the problems being encountered. These problems can be process related, product related, technology related, or personnel related. The faculty will not necessarily answer questions concerning problems being faced, but rather guide the students to come up with their own solution to a problem during discussions in class.

The problems the students face while applying the process will lead to the identification of process weaknesses. The Identify Process Weaknesses phase is a time when students actively look for problems in the existing process and record them. During class time the students will be encouraged to identify process weaknesses and evaluate the impact they have had on the products they have produced. The students will then be encouraged to categorize these weaknesses by severity. The faculty will be responsible for guiding these discussions and pointing out problems that may have been overlooked by the students. Once a list of weaknesses has been assembled, it will be submitted to the management for review.

After the management has reviewed the list of weaknesses, they will decide which problems must be fixed and submit the list back to the students. The list of approved changes is the foundation for the Modify Process phase. During this phase the students look for solutions to the process weaknesses that must be fixed. During class time, the faculty will guide discussions on potential solutions presented by individual students and suggest software engineering articles that may provide solutions.

Using this learning process, the students are responsible for producing the materials they will use during the course. They are also responsible for continual process improvement. The faculty are responsible for facilitating
this learning process and consulting with students on project decisions they make. Much of the faculty’s time will be spent with the students facilitating discussion concerning what occurred during the last increment and what solutions (process, product, technology, or personnel) may be used to improve performance on the next increment. We believe this experiential-based learning will provide students with a deeper understanding and appreciation of software engineering.

**Grading**

Software Factory courses are meant to be participatory classes. Management will be responsible for writing performance reviews for each team member, each semester. These performance evaluations will be the basis for the students’ course grades assigned by the faculty member. Managers are better suited to evaluate the team members than the faculty due to their working relationship. By evaluating their team members, managers will gain valuable experience in interpersonal communication skills and writing performance evaluations. The performance review will give managers leverage to motivate the team members.

Managers’ grades will be based on evaluations from team members. This grading method provides checks and balances between management and team members.

**Implementation Alternatives**

Many academic institutions already have existing computer science programs in place. Therefore, it is important to analyze the existing situation and choose an alternative that provides the most benefits with the least amount of disruption to the institution and students. We have considered various approaches for implementing the Software Factory. These approaches fall into three basic categories: incremental, big bang, and sandwich. With each approach, the institutional environment as well as potential process models and workflow should be considered.

**Incremental Approach**

The incremental approach is defined as introducing the Software Factory in incremental stages, either bottom–up or top–down. With the bottom–up incremental approach, the implementation of the program starts with the first-year factory courses (Software Development Process, Tools and Testing I & II). With the top–down incremental approach, the fourth-year factory courses
(Software Project Management I & II) are offered in the first year of implementation. In each additional year, the next set of classes is offered until all of the courses for the Software Factory are offered.

With the bottom–up incremental approach it would be difficult, but possible, to support the waterfall and evolutionary process models. Since all the roles of the development organization are not filled until the fourth year, external support would be needed to support the flow of work. The top–down incremental approach lends itself easily to a waterfall process model. In the first year, management would define the processes and environment to be used in the development organization. They also would choose and define the product(s) to be developed. The addition of courses in the following years flows naturally from the waterfall model. Supporting the evolutionary process model while using a top–down incremental approach would be difficult since not all roles of the development organization would be filled until the fourth year of implementation.

Some advantages of implementing the Software Factory courses incrementally include a reduced course load and the possibility of smaller, initial lab space requirements. With either incremental approach, only two additional courses are added each year instead of the full eight courses required for a full implementation of the Software Factory. Additionally, since only the first- or fourth-year students would be involved initially, the institution could incrementally allocate resources and equipment to set up the laboratory space.

Although there are some advantages to an incremental approach, there are also several disadvantages. The most serious concern is the amount of time it will take to be able to evaluate the full benefits and drawbacks of having the Software Factory. An incremental approach would require 4 years to fully staff the development organization. This type of approach also delays using the Software Factory for experimentation and research purposes. A more practical issue that exists with the bottom–up incremental approach is that although only two courses are introduced, the work products for the first-year students to use for the testing aspect of the courses would have to be developed or obtained. And, with each additional year, the work products for all of the courses that have not yet been introduced would have to be developed or obtained. These work products, most likely, would not be used by future generations of the Software Factory, lessening the potential benefits for students.

Implementing the top–down incremental approach starting with the fourth-year students and working backward to the first-year students eliminates some of the problems of the bottom–up incremental approach. Starting with the
management courses allows the work products created during the first year of implementation to be used in the following years. The top–down incremental approach eliminates the need for developing and obtaining external work products to be used by the development organization. Using student-developed work products allows the students to appreciate the effects of their decisions, and provides a more realistic experience. However, implementing the top–down incremental approach might be difficult for institutions with an existing program in place since it would be difficult to impose new requirements on students in the fourth year.

**Big Bang Approach**

In many ways, the big bang approach is the opposite of an incremental approach. In this case, all 4 years of Software Factory courses are offered immediately. Because all roles of the development organization are filled immediately, this approach allows for both the waterfall and evolutionary process models.

With this approach and a waterfall process model, the only work product that would have to be developed would be a design document for the developers to use for the first semester. Since many institutions already offer courses in which a design document is used or developed, these existing documents could be used for the developers in the first year. During the first year, the students will be producing the work products required for the subsequent courses.

Using the big bang approach with an evolutionary process model would not require any external work products. The faculty consultant would need to have a development process and environment established. Then, since all roles exist, the iterations of the evolutionary process model could begin immediately.

Introducing all the new Software Factory courses has several advantages over an incremental approach. This approach provides quick feedback since the Software Factory will be staffed at all levels in the first year. This approach also makes the Software Factory available for experimentation and research more quickly than other approaches.

There are some drawbacks to this approach. The most serious is that introducing eight new courses may be difficult for the existing faculty to cover. One solution would be to rely on adjuncts and lecturers to teach some of the new courses. The big bang approach also suffers from the same problem as the top–down incremental approach in that it assumes that current students will want to participate in the Software Factory. Typically, an institution cannot impose new requirements on existing students, however, some of these students will have to participate in order to staff the Software Factory fully.
Sandwich Approach

A sandwich approach is a compromise between incremental and big bang approaches. With this approach, in the first year of implementation, only the first- and the fourth-year courses are introduced. In the second year, the full set of Software Factory courses is offered. Since management roles exist in the first year, a waterfall process model would work well with this approach. It would be difficult to support an evolutionary process model during the first year of implementation since all roles are not filled until the second year.

A sandwich approach, while reducing the initial new course load that is required by a big bang approach (i.e., four courses instead of eight), still has many of the problems of the bottom–up incremental approach. Since there are no developers in the first year of the Software Factory with this approach, a work product for the testers will have to be developed or obtained by the faculty. In the second year, the faculty would have to develop or provide design documents for the developers. Like a bottom–up incremental approach, having external work products would lessen the experience since these work products may not be used in future years of the Software Factory. A sandwich approach reduces by half the time to have a fully operational Software Factory when compared to an incremental approach, but it would still take two full years.

Other Approaches

Other course combinations requiring 2 years for full implementation have been considered. An inside-out approach where the second- (Software Development and Maintenance I & II) and third- (Software Requirements, Test Planning and Design I & II) year courses are offered in the first year of implementation lends itself easily to an evolutionary process model. In this case, the faculty consultant would act as management and the developers would be asked to take on the role of testers during the first year of implementation. An approach that offers the first- and third-year courses is another interesting option. This approach lends itself more easily to a waterfall process model than to an evolutionary process model. Again, the faculty consultant would fill the role of management during the first year. However, with this option and the waterfall process model, it would not be necessary to have developers until the second year. Finally, an approach that offers the third- and fourth-year courses in the first year of implementation would offer similar advantages as a top–down incremental approach with the added benefit of reducing the total implementation time to 2 years. With this
approach, in the first year, initial management processes are defined; initial projects are selected; and requirements and design begin for the initial projects. This approach follows a waterfall-like process model in the first year.

**Implementation Plan for the Catholic University of America**

The Catholic University of America (CUA) is a small, private university that uses the semester system. There is an existing computer science program in place. The enrollment goal of the computer science program is to have 25 new, first-year students each September. Once this goal is attained, the Software Factory will have five teams, each consisting of 20 students. These numbers do not take into account attrition; however, a small attrition rate should not greatly impact the success of a team. Attrition is one of the biggest problems in industry. It is another real world problem that the students must face.

After considering these alternatives and the existing environment, the big bang approach was chosen by the Catholic University program. Since Catholic University already has an existing program in place, some of the current students will need to incorporate the new courses into their existing programs. To coordinate with the existing programs, the Software Factory courses will fulfill current computer science electives. Students entering into the Software Factory course sequence will enter at a level based on a recommendation from their academic advisor. Transfer students will be handled in the same way as existing, computer science students. All incoming, first-year computer science students will be required to follow the program described in Appendix A.

**EVALUATION PLAN**

We plan to use our objectives in the INTRODUCTION to guide our evaluation plan. Based on our objectives, we would like to answer the following questions:

**Meet the Needs of Industry**

1. Is the industry interested in collaborating with the Software Factory?

   We will keep track of the number of inquiries from the industry. We will monitor the number of projects and experiments completed for the industry in the Software Factory.
Create an Accredited Program

1. Does the new curriculum satisfy ABET curriculum requirements for accreditation?

   We have mapped the courses for the new curriculum to the ABET curriculum requirements. The Software Factory curriculum meets these requirements (see Appendix B).

2. Is industry happy with our graduates?

   To address this question, we need to get feedback from employers and potential employers of our graduates.

3. Are our graduates trained in technologies that the industry is interested in?

   To address this question, we will compare technologies used in the Software Factory with those that are in demand in industry.

Attract and Retain Quality Students

1. Has the number of applicants in computer science increased?

   We will look at the total number of computer science applicants each year over the 5 years prior to the introduction of the Software Factory and compare it to the total number of computer science applicants each year since the introduction of the Software Factory.

2. Have the GPA and SAT scores of applicants in computer science increased?

   We will look at the GPA, SAT scores, and quality ratings of computer science students for the 5 years prior to the introduction of the Software Factory. We will compare these scores and ratings to the GPA, SAT scores, and quality ratings of applicants since the introduction of the Software Factory.

3. Has the retention rate been maintained or improved?

   We will look at the retention rate of computer science students for the 5 years prior to the introduction of the Software Factory. We will look for the same or better rate of retention in each year since the introduction of the new curriculum.
Conduct Empirical Software Engineering Research

1. Has the factory been used to conduct empirical software engineering research?

The number of experiments completed in the Software Factory will be monitored. We will also keep track of the number of papers generated from work done in the factory.

Encourage Multidisciplinary Collaboration

1. Are other departments interested in the factory?

We will keep track of the number of grants supporting the factory, the dollar amount of funding for the factory and the number of departments utilizing the factory.

SUMMARY AND FUTURE WORK

With current, undergraduate, computer science curricula, students graduate with technical skills, but lack practical software engineering skills needed in industry. We believe the Software Factory concept will benefit students, faculty, the academic institution, and the industry. Students will learn more and retain more by putting their newly acquired skills to use in a real software development organization. The faculty will have the opportunity to contract out software to students running the Software Factory and conduct software engineering experimentation. The academic institution will be able to attract a greater number of quality students. Finally, the industry will benefit by gaining access to students who have experience in a real world, software development organization upon graduation.

The new curriculum was approved by the Department of Electrical Engineering and Computer Science at CUA in the spring of 2000. Currently, we are helping other academic institutions implement the Software Factory curriculum. Washington College, a small, private college in Chestertown, Maryland will be offering the Software Factory curriculum in September 2001 under the name of the Software Studio. Washington College has chosen to use the 2-year, top-down, incremental implementation approach. At Washington College, the Software Studio will begin with a waterfall-like process model and move to an evolutionary one. We will be active participants in the development and evaluation of this curriculum as well.
REFERENCES


APPENDIX A

Course Sequence/Description

1st Semester:
xxx: Required Elective
xxx: Required Elective
MATH 121: Calculus I (4)
CSC 131: Computer Science I (Java) (3)
CSC 151: Software Development Process, Tools, and Testing I (3)

2nd Semester:
xxx: Required Elective
xxx: Required Elective
MATH 122: Calculus II (4)
CSC 132: Computer Science II (Java) (3)
CSC 152: Software Development Process, Tools, and Testing II (3)

3rd Semester:
xxx: Required Elective
xxx: Required Elective
CSC 211: Discrete Structures (3)
CSC 231: Data Structures (3)
CSC 251: Software Development and Maintenance I (Code and Unit Test) (3)

4th Semester:
xxx: Required Elective
xxx: Required Elective
CSC 212: Theory of Computation (3)
CSC 222: Computer Organization and Assembly Language (3)
CSC 252: Software Development and Maintenance II (Code and Unit Test) (3)

5th Semester:
xxx: Required Elective
MATH 501: Linear Algebra (3)
CSC 311: Design and Analysis of Algorithms (3)
CSC 331: Programming Languages (3)
CSC 351: Software Requirements, Test Planning, and Design I (3)

6th Semester:
xxx: Required Elective
MATH 531: Probability and Statistics (3)
CSC xxx: Computer Science Elective (3)
CSC xxx: Computer Science Elective (3)
CSC 352: Software Requirements, Test Planning, and Design II (3)

7th Semester:
xxx: Required Elective
xxx: Required Elective
CSC xxx: Computer Science Elective (3)
CSC xxx: Computer Science Elective (3)
CSC 451: Software Project Management I (3)

8th Semester:
xxx: Required Elective
xxx: Required Elective
CSC xxx: Computer Science Elective (3)
CSC xxx: Computer Science Elective (3)
CSC 452: Software Project Management II (3)

Required Electives: (44)

Science:
SCI xxx: 1st semester of a lab course (4)
SCI xxx: 2nd semester of a lab course (4)
SCI xxx: Science Elective (3)
SCI xxx: Science Elective (3)

English:
ENG 101 (or 105,103,104): English Composition (3)

Religion:
REL 201 (or 203): Christian Difference (201 cannot be taken 1st semester of first year) (3)
REL xxx: Religion Elective (3)
REL xxx: Religion Elective (3)

Philosophy:
PHIL 362: Professional Ethics in Engineering (3)

Liberal Studies:
LS xxx: Liberal Studies Elective (3)
LS xxx: Liberal Studies Elective (3)
LS xxx: Liberal Studies Elective (3)
LS xxx: Liberal Studies Elective (3)
LS xxx: Liberal Studies Elective (3)

Additional Requirements

1. To be accepted as a major, a student must have completed CSC 131, CSC 132, CSC 211, and CSC 231 with a minimum GPA of 2.5 in these four courses.

2. To ensure competence in the core material, all core mathematics and computer science courses must be passed with a grade of C or better to satisfy the requirements of the degree.

3. To ensure breadth in the choice of the six computer science electives, at least one course, and no more than two courses, must be taken from each of the following areas: Computer Science Foundations (CSC x1x), Computer Systems (CSC x2x), Software Systems (CSC x3x), and Computing Methodologies (CSC x4x).
Mapping to ABET Curriculum Requirements

The table below contains section IV from the ABET requirements for accreditation. The standards are given on the left and the evidence of the Software Factory curriculum on the right.

### IV. Curriculum

Intent: The curriculum is consistent with program’s documented objectives. It combines technical requirements with general education requirements and electives to prepare students for a professional career in the computer field, for further study in computer science, and for functioning in modern society. The technical requirements include up-to-date coverage of basic and advanced topics in computer science as well as an emphasis on science and mathematics.

<table>
<thead>
<tr>
<th>Standards: General</th>
<th>Evidence:</th>
</tr>
</thead>
<tbody>
<tr>
<td>IV-1. The curriculum must include at least 40 semester hours of up-to-date study in computer science topics.</td>
<td>63 semester hours</td>
</tr>
<tr>
<td>CSC 131: Computer Science I (Java) (3)</td>
<td></td>
</tr>
<tr>
<td>CSC 151: Software Development Process, Tools, and Testing I (3)</td>
<td></td>
</tr>
<tr>
<td>CSC 132: Computer Science II (Java) (3)</td>
<td></td>
</tr>
<tr>
<td>CSC 152: Software Development Process, Tools, and Testing II (3)</td>
<td></td>
</tr>
<tr>
<td>CSC 231: Data Structures (3)</td>
<td></td>
</tr>
<tr>
<td>CSC 251: Software Development and Maintenance I (Code and Unit Test) (3)</td>
<td></td>
</tr>
<tr>
<td>CSC 212: Theory of Computation (3)</td>
<td></td>
</tr>
<tr>
<td>CSC 222: Computer Organization and Assembly Language (3)</td>
<td></td>
</tr>
<tr>
<td>CSC 252: Software Development and Maintenance II (Code and Unit Test) (3)</td>
<td></td>
</tr>
</tbody>
</table>
IV-2. The curriculum must contain at least 30 semester hours of study in mathematics and science as specified below under Mathematics and Science.

CSC 311: Design and Analysis of Algorithms (3)
CSC 331: Programming Languages (3)
CSC 351: Software Requirements, Test Planning, and Design I (3)
CSC xxx: Computer Science Elective (3)
CSC xxx: Computer Science Elective (3)
CSC 352: Software Requirements, Test Planning, and Design II (3)
CSC xxx: Computer Science Elective (3)
CSC xxx: Computer Science Elective (3)
CSC 451: Software Project Management I (3)
CSC xxx: Computer Science Elective (3)
CSC xxx: Computer Science Elective (3)
CSC 452: Software Project Management II (3)

31 semester hours
Mathematics:
MATH 121: Calculus I (4)
MATH 122: Calculus II (4)
CSC 211: Discrete Structures (3)
MATH 501: Linear Algebra (3)
MATH 531: Probability and Statistics (3)
Science:
SCI xxx: 1st semester of a lab course (4)
SCI xxx: 2nd semester of a lab course (4)
SCI xxx: Science Elective (3)
SCI xxx: Science Elective (3)

IV-3. The curriculum must include at least 30 semester hours of study in humanities, social sciences, arts and other disciplines that serve to broaden the background of the student.

30 semester hours
English:
ENG 101 (or 105, 103, 104): English Composition (3)
Religion:
REL 201 (or 203): Christian Difference (201 cannot be taken 1st semester of first year) (3)
REL xxx: Religion Elective (3)
REL xxx: Religion Elective (3)

Philosophy:
PHIL 362: Professional Ethics in Engineering (3)

Liberal Studies:
LS xxx: Liberal Studies Elective (3)
LS xxx: Liberal Studies Elective (3)
LS xxx: Liberal Studies Elective (3)
LS xxx: Liberal Studies Elective (3)
LS xxx: Liberal Studies Elective (3)

IV-4. The curriculum must be consistent with the documented objectives of the program.

Computer Science

IV-5. All students must take a broad-based core of fundamental computer science material consisting of at least 16 semester hours.

45 semester hours
CSC 131: Computer Science I (Java) (3)
CSC 151: Software Development Process, Tools, and Testing I (3)
CSC 132: Computer Science II (Java) (3)
CSC 152: Software Development Process, Tools, and Testing II (3)
CSC 231: Data Structures (3)
CSC 251: Software Development and Maintenance I (Code and Unit Test) (3)
CSC 212: Theory of Computation (3)
CSC 222: Computer Organization and Assembly Language (3)
CSC 252: Software Development and Maintenance II (Code and Unit Test) (3)
CSC 311: Design and Analysis of Algorithms (3)
CSC 331: Programming Languages (3)
CSC 351: Software Requirements, Test Planning, and Design I (3)
CSC 352: Software Requirements, Test Planning, and Design II (3)
CSC 451: Software Project Management I (3)
CSC 452: Software Project Management II (3)
IV-6. The core materials must provide basic coverage of algorithms, data structures, software design, concepts of programming languages, and computer organization and architecture.

IV-7. Theoretical foundations, problem analysis, and solution design must be stressed within the program’s core materials.

IV-8. Students must be exposed to a variety of programming languages and systems and must become proficient in at least one higher level language.

IV-9. All students must take at least 16 semester hours of advanced course work in computer science that provides breadth and builds on the core to provide depth.

Mathematics and Science

IV-10. The curriculum must include at least 15 semester hours of mathematics.

17 semester hours

- MATH 121: Calculus I (4)
- MATH 122: Calculus II (4)
- CSC 211: Discrete Structures (3)
IV-11. Course work in mathematics must include discrete mathematics, differential and integral calculus, and probability and statistics.

IV-12. The curriculum must include at least 12 semester hours of science.

IV-13. Course work in science must include the equivalent of a two-semester sequence in a laboratory science for science or engineering majors.

IV-14. Science course work additional to that specified in Standard IV-13 must be in science courses or courses that enhance the student's ability to apply the scientific method.

Additional Areas of Study

IV-15. The oral communications skills of the student must be developed and applied in the program.

IV-16. The written communications skills of the student must be developed and applied in the program.

IV-17. There must be sufficient coverage of social and ethical implications of computing to give students an understanding of a broad range of issues in this area.