Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 153 (2019) 71–79

Procedia Computer Science

www.elsevier.com/locate/procedia

17th Annual Conference on Systems Engineering Research (CSER)

Educating I-Shaped Computer Science Students to Become T-Shaped System Engineers

Barry Boehm[a,*], Supannika Koolmonojwong[b]

[a]*518 Adelaide Drive, Santa Monica, CA, 90402, USA*
[b]*1320 3rd St., Santa Monica, CA, 90402, USA*

Abstract

With every passing day, software becomes more and more important to the success of the artifacts that we make, sell, buy, use, and evolve. Software increasingly provides a competitive differentiator for products, ways of tailoring them for various uses and users, and ways of fixing or evolving them without expensive product recalls.

Unfortunately, as software becomes more and more ubiquitous and complex, an increasing number of new computer science (CS) courses in web services, big-data analytics, computing security, and machine learning fill up CS students' schedules, leaving little room for non-CS courses providing skills outside of CS. This paper summarizes our experiences in developing and evolving an MS-level software engineering (MSCS-SE) curriculum that takes I-shaped CS BA graduates and enables them to become sufficiently T-shaped to be able to immediately contribute to overall system definition and development on being hired, and to improve their T-shaped skills along their careers.

Section 2 summarizes the primary origins and problems with an I-shaped software workforce. Section 3 describes the origins, development, and evolution of the USC MSCS-Software Engineering program and its foundation-stone, real-client, 2-semester project course. Section 4 elaborates on the team-project course and its mechanisms for strengthening the transition from I-shaped to T-shaped systems thinking. Section 5 provides conclusions.

© 2019 The Authors. Published by Elsevier B.V.
This is an open access article under the CC BY-NC-ND license (https://creativecommons.org/licenses/by-nc-nd/4.0/)
Peer-review under responsibility of the scientific committee of the 17th Annual Conference on Systems Engineering Research (CSER).

* Corresponding author. Tel.: 2137408163; fax: 2137404927.
E-mail address: boehm@usc.edu

1877-0509 © 2019 The Authors. Published by Elsevier B.V.
This is an open access article under the CC BY-NC-ND license (https://creativecommons.org/licenses/by-nc-nd/4.0/)
Peer-review under responsibility of the scientific committee of the 17th Annual Conference on Systems Engineering Research (CSER).
10.1016/j.procs.2019.05.057

Keywords: T-shaped, I-shaped, MS degree, computer science, systems engineering, edducation

1. **Introduction.**

With every passing day, software becomes more and more important to the success of the artifacts that we make, sell, buy, use, and evolve. Software increasingly provides a competitive differentiator for products, ways of tailoring them for various uses and users, and ways of fixing or evolving them without expensive product recalls. Unfortunately, as software becomes more and more ubiquitous and complex, an increasing number of new computer science (CS) courses in web services, big-data analytics, computing security, and machine learning fill up CS students' schedules, leaving little room for non-CS courses providing skills outside of CS. This paper summarizes our experiences in developing and evolving an MS-level software engineering (MSCS-SE) curriculum that takes I-shaped CS BA graduates and enables them to become sufficiently T-shaped to be able to immediately contribute to overall system definition and development on being hired, and to improve their T-shaped skills along their careers. Section 2 summarizes the primary origins and problems with an I-shaped software workforce. Section 3 describes the origins, development, and evolution of the USC MSCS-Software Engineering program and its foundation-stone, real-client, 2-semester project course. Section 4 elaborates on the team-project course and its mechanisms for strengthening the transition from I-shaped to T-shaped systems thinking. Section 5 provides conclusion

2. . Origins of and problems with I-Shaped Software Engineers

When software began to be used for performing functions in a hardware system such as an aircraft, it would usually appear in numerous places at low levels of the aircraft's work breakdown structure. For example, an aircraft has wings as parts, which have ailerons as parts, which have aileron controls as parts, which have sensors as parts, which have sensor software as parts. This caused most software to be developed as isolated small, low level Computer Software Configuration Items, with little understanding of their part in the overall aircraft system. Such narrow foci of the CSCIs owned by other parts of the aircraft hierarchy problems can arise from incompatibilities with the software in atmospheric, propulsion and attitude sensors to accomplish the stabilization objectives [1].

Such discouragement of software engineers to participate in overall system requirements and architecture decisions was exacerbated by software capability models such as the Software Capability Maturity Model [2]. Its first Ability 1 of its first Key Process Area of Requirements Management states, "Analysis and allocation of the system requirements is not the responsibility of the software engineering group but is a prerequisite for their work." Not only did this stimulate more I-shaped software thinking, but also it excluded software technical experts from participating in architectural decisions as software evolved from controlling 8% of an aircraft's capabilities in the 1960s to 80% by the year 2000 [3]. Some other problems created by I-shaped software engineers include:

- The Golden Rule: Do unto others as you would have others do unto you. Programmers often take this literally and create programmer-friendly user interfaces for doctors, nurses, executives, or hardware engineers. A better guideline is the Platinum Rule: Do unto others as they would be done unto.
- Computer scientists prize abstraction, and often undermine the value of prototypes by calling users U1, U2, U3, U4 vs. Jim, Rosa, Ali, and Kathy. We find having the student teams invent Personas to represent classes of stakeholders is both effective and entertaining.
- Making programmer-convenient, but user-inconvenient decisions: an example is creating 10-day data buckets making it easy to program but hard for users to relate these to weekly and monthly planning.

A major problem in cyber-physical-human (CPH) systems is that their engineers are generally not aware of the differences in world-views among hardware, software, and human factors engineers. Table 1 below summarizes some of these differences. Finally, producing specialized, I-shaped practitioners is often reinforced by university research and education priorities: the specialists get the PhDs, the Turing Awards, and the Nobel Prizes. Within academic departments, "breadth" courses are generally within the discipline, not interdisciplinary. Also, many software engineering courses are taught by instructors with little industrial experience, who tend to teach the programming methodology material that they know best.

Table 1. Hardware, Software, and Human Engineer World Views

Difference Area	Hardware/ Physical	Software/Cyber/ Informational	Human Factors
Economies of scale	The more hardware units, the cheaper they are per unit	The more software units, the costlier they are per unit	The larger the team, the less productivity due to communications overhead
Nature of changes	Generally manual, labor-intensive, expensive	Generally straightforward except for software code rot, architecture-breakers	Very good, but dependent on performer knowledge and skills
Incremental development constraints	More inflexible lower limits	More flexible lower limits	Smaller increments easier, if infrequent
Underlying science	Physics, chemistry, continuous mathematics	Discrete mathematics, logic, linguistics	Physiology, behavioral sciences, economics
Testing	By test engineers, much analytic continuity	By test engineers, little analytic continuity	By representative users
Strengths	Creation of physical effects; durability; repeatability; speed of execution; 24/7 operation in wide range of environments; performance monitoring	Low-cost electronic distributed upgrades; flexibility and some adaptability; big-data handling, pattern recognition; multi-tasking and relocatability	Perceiving new patterns; generalization; guiding hypothesis formulation and test; ambiguity resolution; prioritizing during overloads; skills diversity
Weaknesses	Limited flexibility and adaptability; corrosion, wear, stress, fatigue; expensive distributed upgrades; product mismatches; human-developer shortfalls	Complexity, conformity, changeability, invisibility; commonsense reasoning; stress and fatigue effects; product mismatches; human-developer shortfalls	Relatively slow decision making; limited attention, concentration, multitasking, memory recall, environmental conditions; teaming mismatches

3. Origins of Creating the T-Shaped Curriculum

When one of the authors (Boehm) was at Thompson Ramo Wooldridge Inc. (TRW), he had the good fortune to lead an MSCS-SE effort commissioned by Dr. Simon Ramo, the R in TRW and a leading pioneer in systems engineering. It began with a discussion at one of his periodic Electronic Technology Advisory Group meetings, in which several TRW divisions indicated that their business units were being constrained by a shortage of good software people.

Among the resulting initiatives that Dr. Ramo commissioned was an MS-degree work-study program for outstanding undergraduates in computer science and software engineering to be hired by TRW, which he asked Boehm to explore and develop with the major local universities, UCLA and USC. However, he said that the program should not develop pure-software people. He said that the most successful engineers at TRW were T-shaped people, who had strong technical depth in one discipline, but who also had a working knowledge of other disciplines. The resulting proposed program was called the TRW MS Degree in Computer Architecture and Software Engineering.

Initially, we were concerned that UCLA and USC would not be interested in such a program, but at the time they

were getting concerned at the decreasing number of US citizens in their graduate engineering programs. It also helped that Dr. Ramo made a substantial annual contribution to each department's discretionary fund. The program did produce a number of future TRW technical leaders, but the bulk of our software new hires continued to be pure I-shaped computer science graduates.

When Boehm approached early retirement at TRW, he decided that it would be worth a try to increase the number of T-shaped engineering graduates available to companies and public-service organizations. This led to his becoming a professor at USC and initially developing an MS program in software engineering that included courses in user-interaction design, computer hardware-software design, software-system requirements, verification and validation, and management and economics.

Its main core course was a 2-semester team project course, in which the first semester covered project system engineering, in developing operational concepts, requirements, architecture, prototypes, development plans, and generation of evidence that these would be compatible, feasible, and user-satisfactory. Initially each team worked from the same project need statement for which Boehm served as customer: a fire-department dispatching system in 1993-94 and a library selective dissemination of information (SDI) system in 1994-95.

In the fall of 1994, Boehm was surprised by a request for a meeting with several of the USC librarians. They indicated that they had been approached by numerous students in the course asking them for information about how the library worked, and learning about the SDI-system project course. They indicated that the USC Libraries had needs for software applications for which they had insufficient budgets to develop, and wondered if the project course could be modified to enable the student teams to develop such applications, particularly in the area of multimedia archives: student films, fine arts course slides, medical school lung pathology images, medieval manuscripts, early Los Angeles newspapers, business school stock exchange data, etc.

The 1995-96 version of the course was revised to have each student team work with a library client to work out an operational concept, set of requirements, user-interaction prototypes, architecture, development plans, and feasibility evidence for a multimedia archive in the Fall semester, and to develop, verify, validate, and transition the resulting software and operational procedures to the client's organization in the Spring semester [4]..

Over the next 23 years, the real-client, team project course and the MSCS-SE curriculum have evolved through a number of major changes, such as going from programming-intensive to COTS-intensive to cloud services-intensive. The clients are generally non-software people, coming about equally from campus organizations, local-neighborhood small businesses, local community-service organizations, and local government. We have also tailored a counterpart senior-undergraduate capstone project course.

The MS course has produced over 2000 graduates who are considerably more T-shaped than they were when they came. They have also included students from other engineering departments who want to become more familiar with software engineering. They have generally done very well in job interviews, and have generally become corporate assets whose skills have been hard to outsource to India or elsewhere. Several universities have adapted the approach to their programs, aided by assets such as an Electronic Process Guide for applying the approach [5]. The experience has also enabled us to evolve the Incremental Commitment Spiral Model into a mature book and set of practices [6], now used as the course textbook.

4. Role and content of the Foundation-Stone Project Course

4.1 T-Shaped Course Practices

Apart from teaching about foundations and theories on systems and software engineering, it is crucial to provide opportunities to students to practice their T-Shaped skills. The following are several course practices that have been used in the class that contribute to the students' becoming more T-shaped:

- **Visit clients' workplace and jointly develop a desired concept of operation.** These operational concepts include current system workflow and shortfalls, benefits chain identifying initiatives and stakeholders beyond software development, desired capability and workflow improvements, and systems constraints. In this practice,

students also learn and understand about the clients' domain. For our class project clients, roughly 25% of projects each are campus organizations, neighborhood small businesses, local community services, and local government. Students need to understand the differences in their needs, budget, schedule, infrastructure, rules and regulations, and technical resources.

- **Jointly negotiate prioritized stakeholder win-win requirements.** Students and clients have to balance ideal capabilities with available development time and effort, understanding constraints, using a win-win requirements negotiation tool (Winbook) [7] to identify minimum marketable features (MMFs) and to perform early software sizing and costing. During the requirements negotiation, students practice concurrent engineering by concurrently developing and iterating prototypes to clarify system usage, to provide proof of concepts, or to provide feasibility evidence.
- **Jointly develop evaluation criteria for choices of non-developmental items (NDIs)** such as COTS, services, and open source libraries; to assess candidate NDIs and their compatibility; and to converge on a best-fit process. Based on our 25 years experience to date, there are 4 common process patterns currently found in this class: Architected Agile, Single NDI, NDI-intensive, and Services-intensive. Students use a process selection support tool [5] to help in selecting the right process.
- **Jointly determine and prioritize project risks, develop risk mitigation plans.** During the potential client meeting, clients are informed about the core practices used in the class. One of the most important practices is risk management. Students perform weekly risk analysis including continuing risk mitigation progress monitoring and evolution. At the same time, clients acknowledge the risks and provide feedback to the team.
- **Develop clients' business case linking investments to quantitative and qualitative benefits.** As part of the project feasibility analysis, with inputs from clients, students determine added client investments such as database conversion, maintenance, end-user training, system sustainment, potential business growth or income and develop return on investment (ROI) and breakeven analyses.
- **Identify complementary client activities.** Besides software development activities, students have to also plan for transition, deployment, and sustainment. This practice includes developing client transition/cutover plans, creating life cycle support plans, identifying interoperability coordination, and software / hardware upgrades.
- **Participate in 4 major milestone reviews with clients and instructors.** Everyone presents their main contributions, risks, concerns and future plans. Clients and instructors provide feedback and suggestions to the team. All stakeholders commit on action items covering all aspects of the project.
- **Develop initial increment and hold a client Core Capability Drivethrough** (CCD) to validate feasibility, identify emergent requirements, and clarify needed business process re-engineering. CCD is an activity that allows the clients or potential users to have hands-on experience on the system. Compared to a general demo presentation to the clients, with CCD, the clients have a better understanding about the system and can identify further needed actions for the team or themselves.
- **Jointly negotiate prioritized end-game revisions**. Although the class provides a default list of project deliverables, students may choose to tailor the process by negotiating with clients to opt out unnecessary items or opt in additional activities such as added testing, deliverable architecture description, test suites or technical user and maintenance manuals.
- **Transition software and support materials**. Clients are informed about the unavailability of students after the semester is over, hence, students have to plan on knowledge transfer to clients, which usually include training to clients, staff, maintainer, and initial end users. It is also quite often that students are hired part-time to help initial system evolution.

4.2 Software Systems Engineering Course

We decided to offer the course as a foundation-stone 2-semster course that students would begin in their first semester, often in parallel to some of the other courses rather than doing as capstone project course at the end. This gave them a working appreciation of the content of the other courses, as they could see how their content applied to their project experience.

CSCI577ab [8] are the software engineering project courses at the University of Southern California (USC)'s Master of Computer Science with Specialization in Software Engineering (MSCS-SE). The main objective of the courses is

to prepare students for software leadership careers through the 2050's. Software Engineering I or CSCI577a in the Fall semester focuses on software-intensive systems engineering, including system operational concept formulation, requirements negotiation and definition, prototyping, COTS and services evaluation and selection, system and software architecture definition, life cycle plans and processes, risk analysis, feasibility analysis, and verification and validation. Software Engineering II or CSCI577b in the Spring semester focuses on software product implementation, integration, test, documentation, transition, and maintenance with an emphasis on quality software production.

In the course, students work as a team to develop e-services projects for small businesses, local government, campus users, or nonprofit organizations. They perform various systems engineering roles such as Requirements Engineer, System Architect, Operational Concept Engineer, Verification and Validation (V&V) Personnel, and Life Cycle Planner. In addition, students can apply systems engineering practices and produce artifacts, such as Operational Concept Description and System and Software Architecture Description. Moreover, working in teams allows students to develop their projects in real world environment by having client representatives, 5-6 on-campus students (co-located), and 1-2 remote students who are mainly working as professional systems engineers, and perform V&V functions.

As mentioned above, the maturation of COTS products and cloud services have made it possible for many student project teams to deliver essential software and data management capabilities for their small business, community services, and local government services clients in a single semester. This has reduced enrollments in the second semester of the project course, and caused us to drop the second semenster as a required core course for the MS-CS degree in Sostfare Engineering. The current core courses for the degree are now the first semesteer of the project course, Software Management and Economics, and Computer Systems Engineering and Architecture. This also frees up students' choices of the other key courses counting toward the MSCS-SE degree such as Requirements Engineering, Testing and Analysis, Multimedia Systems Design, and Softare for Embedded Systems. This is generally a positive development, as students taking the specalty courses can bring the knowledge to the student projects, and vice versa.

4.3 The Incremental Commitment Spiral Model (ICSM)

The Incremental Commitment Spiral Model (ICSM) is a refined version of the original spiral process model. ICSM covers the full system development life cycle consisting of five incrementally-defined life cycle phases (Exploration, Valuation, Foundations, Development, and Operations phases). The ICSM, as shown in Figure 1, is not a single one-size-fits-all model but a risk-driven framework for tailoring a process that best fits a project's situation by using the risk-based decision options at the end of each spiral phase. The ICSM is currently used as the process model for the software engineering class, as supported by an Eletronic Process Guide (EPG) [5] developed via the Rational Method Composer [9].

The four underlying principles of the ICSM are:

1. **Stakeholder Value-based system definition and evolution** - The project should be developed based on satisfying the value propositions of all success-critical stakeholders. Otherwise, the stakeholders will frequently not commit to their project roles, which will lead to project rejection or major rework.
2. **Incremental commitment and accountability –** Stakeholders do not commit to a single pre-defined set of requirements and resource contributions, but commit incrementally as the nature of the system is better understood. Otherwise, the project often becomes locked into out-of-date concepts of what the system should provide its stakeholders, leading to project rejection or major rework.
3. **Concurrent system and software definition and development** – Contrary to sequential development, the concurrent development of requirements, solutions, hardware, software, and human factors allows the project to move faster, avoid premature commitments, and be more flexible to yield the best results.
4. **Evidence and risk-based decision making** – Making the evidence of project feasibility a first-class deliverable provides a way to synchronize and stabilize the concurrently-defined system elements. Shortfalls in evidence are uncertainties that identify the level of risk of proceeding without stronger evidence of project feasibility.

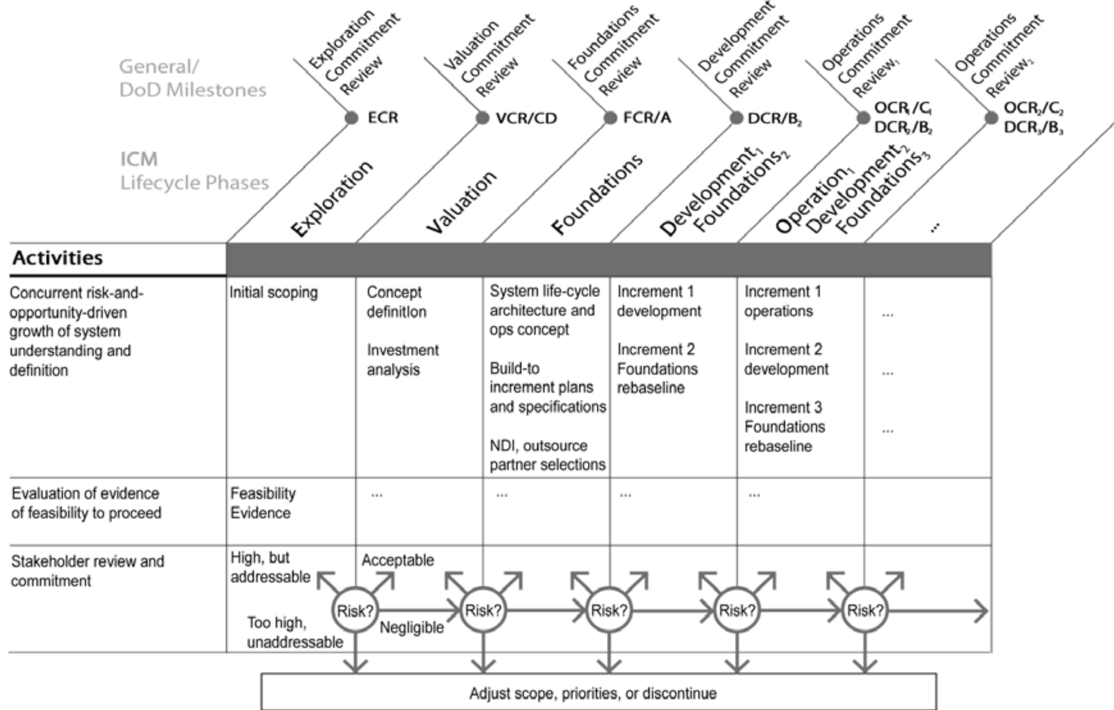


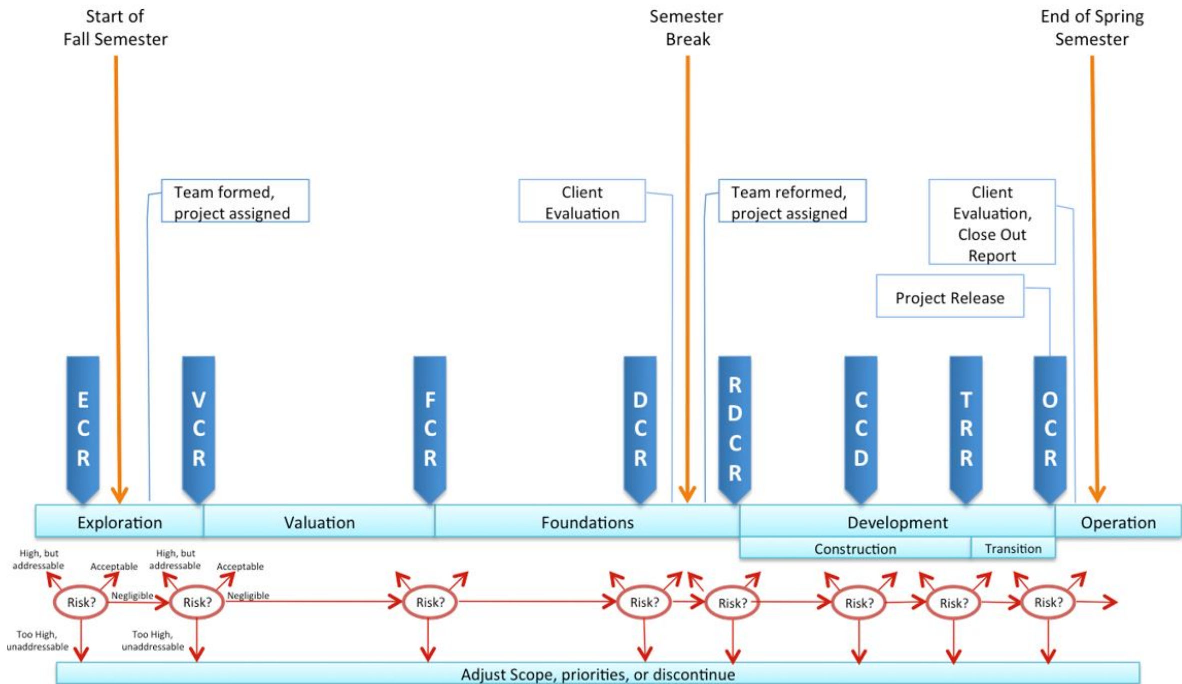
Fig. 1. The Incremental Commitment Spiral Model – Phase View

The best practices based on the four ICSM principles have been applied to the class. A milestone system is used to check the feasibility evidence, analyze risks, stabilize and synchronize the progress, and confirm commitment from stakeholders. As shown in Figure 2, there is 1 minor milestone, Valuation Commitment Review, and 2 major milestones, Foundations Commitment Review and Development Commitment Review in the Fall semester. Later, there are 2 major milestones in the Spring semester, Re-baselined Development Commitment Review and Transition Readiness Review, and 2 minor milestones, Core Capability Drivethrough and Operations Commitment Review. In addition, there are various industrial tools and techniques applied to the class such as configuration management, independent verification and validation, project plan, unified modeling language, feasibility evidence, and business case analysis. Although these projects are small compared to industrial projects and could be overkill for projects at this size, students are provided with an opportunity to build large-system skills.

5. Resulting Benefits

The main benefits are for the students. When they go to job fairs or hiring interviews, they have a portfolio describing the project on which they participated, and can demonstrate their level of T-shaped capability in not only programming, but also capabilities in economics, business case analysis, human factors in prototyping, life cycle maintenance preparation, and domain skills in the domain of their team project. They get better job offers, and the hiring companies subsequently come back looking for more graduates of the MS program. Another main set of benefits come for the clients, who generally receive more capability than they expected, often after one semester rather than two. And for the local charity organizations and local small businesses, the resulting software systems help benefit the local USC community. Other beneficiaries are our Ph.D. students, who have a critical mass of projects on which they can test their hypotheses. For example, they were able to show on the student projects that the value-based

prioritization of features to inspect and test enabled projects to double the value of the results per hour of inspection or testing. And as instructors trying to keep up with the latest in software technology and incorporate it in our courses, the experiences of applying new technologies to practical projects enable us to keep up to date with rapidly-evolving software technology and its effects.



CCD-Core Capability Drivethrough; **DCR**- Development Commitment Review; **ECR**-Evaluation Commitment Review; **FCR**-Foundations Commitment Review; **OCR**- Operational Commitment Review; **RDCR**-Rebaselined Development Commitment Review; **TRR**-Transition Readiness Review; **VCR**-Valuation Commitment Review

iew
CCD-Core Capability Drivethrough; **DCR**- Development Commitment Review; **ECR**-Evaluation Commitment Review; **FCR**-Foundations Commitment Review; **OCR**- Operational Commitment Review; **RDCR**-Rebaselined Development Commitment Review; **TRR**-Transition Readiness Review; **VCR**-Valuation Commitment Review

Fig.2. Software Engineering Class Timeline in ICSM EPG

6. Conclusions

The students that we are educating today will have careers extending into the 2050s. It is hard to imagine what their workdays will look like, but it is highly likely that they will be collaborating with experts in other disciplines to create cyber-physical-human systems that improve the lives of their users. Even today, it is increasingly important for software engineers to become more T-shaped to be able to help develop and evolve cyber-physical systems representative of the future, such as internets of things, social networking applications, and more complex systems of systems, all undergoing increasingly rapid change.

Our experience in developing, annually applying and evaluating, and evolving the curriculum and project course guidelines and infrastructure across 25 years of technology and personnel changes, has shown us that the approach is sustainable, but requires considerable effort to evolve. We need to keep the infrastructure stable while the projects are going on, but our summers are busy rebaselining the infrastructure and tools to accommodate the feedback in the client evaluations, student critiques, and project reviews.

So far, we have been able to accommodate paradigm shifts from programming-intensive to COTS-intensive to cloud

services-intensive applications; from pure plan-driven to varying mixes of plan-driven and agile development; and from desktop to mobile and Internet of Things applications. The Electronic Process Guide has made the evolution much more achievable. Again, overall, this experience also requires us to keep at the frontier of software engineering practice, and to continue to satisfy clients, students, and hiring managers in providing the benefits of having more T-shaped, system-thinking software engineers.

References

[1] M. Maier, "System and Software Architecture Reconciliation," Systems Engineering 9 (2), 2006, pp. 146-159.

[2] M. Paulk, C. Weber, B. Curtis, and M. Chrissis, The Capability Maturity Model: Guidelines for Improving the Software Process, Addison Wesley, 1994.

[3] J. Ferguson, "Crouching Dragon, Hidden Software: Software in DOD Weapon Systems," IEEE Software 18(4), 2001, pp. 105-107.

[4] B. Boehm, A. Egyed, D. Port, A. Shah, J. Kwan, R. Madachy, "Using the WinWin Spiral Model: A Case Study," IEEE Computer, Volume 31, Number 7, July 1998, pp. 33-44.

[5] S. Koolmanojwong, and B. Boehm, "Educating Software Engineers to Become Systems Engineers", Proceedings of the 24th Conference on Software Engineering Education and Training - CSEET, Waikiki, HI, 2011.

[6] B. Boehm, J. Lane, S. Koolmanojwong, and R. Turner, The Incremental Commitment Spiral Model, Addison Wesley, 2014.

[7] N. Kukreja and B. Boehm, Process Implications of Social Networking-Based Requirements Negotiation Tools, In International Conference on System and Software Process (ICSSP) 2012

[8] CSCI577 – Software Engineering class – http://www.greenbay.usc/edu

[9] P. Haumer, IBM Rational Method Composer: Part 1: Key concepts, December 2005, http://www.ibm.com/developerworks/rational/library/dec05/haumer/.