Empirical Computational Thinking

Philip M. Johnson
Collaborative Software Development Laboratory
Department of Information and Computer Sciences
University of Hawai'i
Honolulu, HI 96822
johnson@hawaii.edu

May 8, 2009

Abstract

This technical report presents an edited version of a proposal to the NSF CPATH program. The vision of this proposal is to develop and institutionalize a new approach to computational thinking where abstraction and automation combine to transform the use of *empirical thinking* in software development. We call this approach "empirical computational thinking", or *e*CT. The goal of this research is to explore, evaluate, and institutionalize techniques and technologies for *e*CT, building upon research and education by ourselves and others in empirically-based software development.

1 Project Vision, Goals, Objectives, and Outcomes

Jeannette Wing writes, "Computational thinking involves solving problems, designing systems, and understanding human behavior, by drawing on the concepts fundamental to computer science" (Wing, 2006). In her presentation "Computational Thinking and Thinking About Computation", Wing refines her view of these fundamental computer science concepts in terms of the "Two As": Abstraction and Automation. Activities related to abstraction involve choosing the right abstractions, operating at multiple levels of abstraction, and defining relationships between abstractions. Activities related to automation involve mechanizing abstraction via precise notations and models. In essence, automation amplifies the power of abstraction. Computational thinking, from this perspective, involves the correct choice of abstraction combined with the correct choice of automation.

The vision of this proposal is to develop and institutionalize a new approach to computational thinking where abstraction and automation combine to transform the use of *empirical thinking* in software development. We call this approach "empirical computational thinking", or *e*CT.

To introduce our approach, we must first address what is meant by empirical thinking. The term "empirical" is variously defined as "derived from experiment and observation rather than theory";

"evidence or consequences observable by the senses"; and "capable of being verified or disproved by observation or experiment."

Given these definitions, it is clear that some degree of empirical thinking is already commonplace in software development. For example, beginning programmers use empirical thinking when they "observe" the output of the compiler to learn how to write syntactically correct programs. Beginners also tend to make extensive use of "experimentation": they execute their program with example data, compare the actual behavior to what they expect, then make modifications until the observed behavior matches their expectations.

These examples of empirical thinking, while typical for beginning programmers, do not scale well because they lack both abstraction and automation. Thus, they fail to constitute the kind of computational thinking of interest to the CPATH program, and they fail as well to be *e*CT.

One would hope that as students progress into more advanced software development courses, the curriculum would scale in at least two ways. First, the complexity, size, and number of people involved in a software development project would scale upwards. Second, the level of abstraction and automation in their empirical thinking would scale commensurately. Unfortunately, while advanced software development courses certainly require students to develop significantly more sophisticated systems than their introductory counterparts, the use of empirical thinking remains mostly non-abstract and non-automated. The principle computational support for advanced programming classes is an integrated development environment such as Eclipse or Visual Studio. While this is a significant advance over vanilla text editors, such IDEs provide relatively little in the way of abstraction or automation for empirical thinking about the products and processes of software development.

Supporting abstraction in empirical thinking for software development generally means creating quantitative models for important development concepts. For example, test quality is an important concept that is commonly emphasized in advanced software development courses. One quantitative model for test quality is line-level test coverage, which is generally expressed as the percentage of source lines of code in the software exercised by the test cases. Another important concept is complexity, and quantitative models such as afferent and efferent coupling or cyclomatic complexity provide abstract, empirical representations for this concept. Some aspects of design quality can be observed through tools that, for example, generate UML representations of the source code and provide rule-based critiques. Even "agile" concepts such as "commit early, commit often" or "collective code ownership" can support abstract, empirical models. For example, "commit early" can be modeled as the percentage of files in the system that are committed within a certain number of days of their creation. "Collective code ownership" can be modeled by the percentage of files in the system that have been edited by every member of the development team.

Supporting automation for these abstractions of empirical thinking for software development means tool support for collecting, analyzing, disseminating, and interpreting these abstractions. For example, an automated process can run once a day and calculate the current coverage and complexity values for the system. These values can be made available to the user by a web application. Alternatively, email or Twitter "alerts" can be sent to the developers when coverage crosses a threshold and becomes too low, or coupling crosses a threshold and becomes too high. Plugins to development tools like IDEs can collect information on which files are edited when in order to determine the age of a file when it is first committed, or the degree of collective editing on the file.

Thus, our vision for *e*CT includes programming as an activity that is rich in automated, abstract

representations of development processes and products, made available conveniently and appropriately for observation and reflection by the programmers. It also includes education in the analytic capabilities required to effectively interpret these representations, to understand their limitations as representations of reality, to know when to take action based upon them and what kind of action is warranted.

The goal of this research is to explore, evaluate, and institutionalize techniques and technologies for *e*CT, building upon research and education by ourselves and others in empirically-based software development. For example, we recently performed an initial evaluation of a novel system and associated curriculum we developed called the "Software Intensive Care Unit" (Johnson and Zhang, 2009). In this approach, sensors attached to development tools automatically collect student process and product data and abstract it into a set of ten "vital signs" that provide an empirical basis for students to assess the "health" of their ongoing projects. The Software ICU is an example of *e*CT, as it supports both automated and abstract empirical thinking about the current state and past history of both their projects and their group processes. Our curriculum materials taught students how to introduce the Software ICU data collection sensors into their laptop development environments, how to obtain the analyses, and how to interpret the results. Section 3 provides more details on our own research and educational initiatives.

While we are excited by the potential of our own prior work in *e*CT, Section 2 overviews other research and educational initiatives that also conform to our vision for *e*CT, such as PSP (Humphrey, 1995), SimSE (Navarro and van der Hoek, 2007), and Win-Win (Valerdi and Madachy, 2007). We intend to organize and develop a constellation of approaches to *e*CT and build a body of knowledge that enables future researchers and educators to understand the comparative strengths and weaknesses of the various approaches and to create innovative new approaches to *e*CT that synthesize and/or extend present day capabilities.

To achieve this goal, we will pursue a variety of objectives, as detailed in Section 4. We will develop the Common *e*CT Evaluation Framework, which will provide an efficient and effective mechanism for eliciting useful information about individual *e*CT initiatives. We will create new curriculum at the University of Hawaii that builds upon our technological and pedagogical innovations of the past ten years, and that supports early evaluation of common facilities like the evaluation framework. We will create a repository of curriculum development materials as well as a repository of public outcome data. We will utilize social networking technologies such as Facebook and LinkIn to create long-term connections with *e*CT participants and enable follow-up research on their *e*CT experience.

2 Intellectual Basis/Related Work

In 1995, Watts Humphrey authored *A Discipline for Software Engineering*, a ground-breaking text that adapted organizational-level software measurement and analysis techniques to the individual developer along with a one semester curriculum (Humphrey, 1995). These techniques are called the Personal Software Process (PSP), and form the basis for the Team Software Process (TSP), which extends the method to groups of developers.

The PSP is the best known and most widespread approach to empirical thinking in the advanced software development curriculum (Maletic et al., 2001; Abrahamsson and Kautz, 2002; Lisack, 2000; Carrington et al., 2001; Ceberio-Verghese, 1996; Borstler et al., 2002). The ap-

proach requires students to develop a series of software projects, typically six to eight during a single semester. Both process and product measures are gathered about each project, and the measurements become increasingly detailed as the semester proceeds. After the first three projects are completed, the students can use the completed projects as historical data to support quality improvement (by identifying repeated types of defects) and estimation (through simple linear regression). The PSP and TSP enjoy strong support from the Software Engineering Institute, which has published a number of case studies indicating success in a classroom setting and which sponsors a yearly symposium to publicize academic and industry experiences (Ferguson et al., 1997; Hayes and Over, 1997). The PSP/TSP enable support for very basic levels of abstraction and automation of empirical thinking. For example, the PSP Dashboard, Jasmine, and LEAP are tools that allow students to enter the data they collect and that automate the calculation of regression lines.

Conn developed a metrics-based software engineering course called the IS Integrated Capstone Project (Conn, 2004). The metrics were closely aligned with the PSP/TSP format, though some of the process constraints were relaxed.

Robillard designed a project-based course in which advanced undergraduates were required to fill out logs that specified the time spent on various activities (Robillard, 1998). However, minimal abstraction and no automation was supported.

Boehm employs the Cocomo cost modeling framework to provide advanced undergraduates with empirical feedback about the costs and required resources for their projects (Valerdi and Madachy, 2007).

Jaccheri has designed courses on empirical software engineering at both the undergraduate and graduate levels (Dingsoyr et al., 1999; Jaccheri and Osterlie, 2005). The undergraduate course revolved around process improvement experiences, while the graduate course focused on empirical research methods.

A number of researchers have explored providing advanced undergraduates with observational data about software development practices through simulation. For example, the SimSE environment (Navarro and van der Hoek, 2007, 2009) allows empirical observation of six different processes, including a waterfall model, incremental model, XP model, code inspection model, and so forth. SESAM (Drappa and Ludewig, 2000) is a textual simulation in which the student manages a project via commands such as "Start preparing the specification" and receives feedback from the system such as "During testing, I have detected bugs". The SimVBSE environment teaches value-based software engineering by simulating the various stakeholders and their needs (Jain and Boehm, 2006).

The Retina system automatically collects editing and compilation data on beginning programmers, which it then abstracts using a recommendation and suggestion subsystem (Murphy et al., 2009). Retina can notice, for example, when a student is getting many more errors per compilation than other students in the class, and recommend that the student might want to break the work down into smaller pieces. Retina demonstrates that there is potential for the use *e*CT throughout the software development curriculum.

An important benefit of *e*CT is that it provides students with a firm foundation for *scientific* and *evidence-based* thinking.

John Dewey provides one of the earliest, and most eloquent descriptions of the difference between empirical and scientific thinking (Dewey, 1910). In his chapter "Empirical and Scientific Thinking", Dewey begins by noting that empirical thinking, which is based purely on observa-

tion, has been used by humans throughout history as an effective way of understanding through association.

A modern example of empirical thinking involves the swarms of poisonous box jellyfish that periodically invade Waikiki Beach in Hawaii. It was discovered by a lifeguard in the 1970's that their appearance is correlated with the lunar cycle: approximately 7-11 days after the full moon, the jellyfish will appear for approximately three days. As there is no theory explaining how or why this correlation exists, it is an example of purely empirical thinking. Nevertheless, it is both accurate and useful, and Hawaii radio and TV all broadcast warnings to beach goers based upon this association.

Although the above example shows that empirical thinking can be both accurate and useful, Dewey explains that there can be dangers if correlation is confused with causality. To address this problem, he introduces the scientific method. From Dewey's point of view, the scientific method involves active experimentation under controlled or semi-controlled conditions (as opposed to passive observation) and the formation of testable theories that introduce causal mechanisms (for which evidence can be gathered to support, refute, or refine).

A related effort is the application of evidence-based medical research techniques to software development (Kitchenham et al., 2004; Kitchenham, 2004), which involves a five step method: (1) Convert the need for information [about a software engineering practice] into an answerable question; (2) Track down the best evidence available for answering the question; (3) Critically appraise that evidence using systematic review for its validity (closeness to the truth), impact (size of the effect), and applicability (usefulness in software development practice); (4) Integrate the critical appraisal with current software engineering knowledge and stakeholder values [to support decision-making]; (5) Evaluate the effectiveness and efficiency in applying Steps 1-4 and seek ways to improve them for next time.

3 Current State

For over ten years, we have been exploring empirical software engineering techniques and their applications in the classroom setting as part of our research in the Collaborative Software Development Laboratory at the University of Hawaii. This section summarizes our prior studies and the current state of practice in our institution.

PSP. Beginning in the late 1990's, we instituted the use of the Personal Software Process in both undergraduate and graduate software engineering courses. While our outcomes were quite positive and in line with the data gathered by the Software Engineering Institute, we were concerned by the possibility of data quality problems and the lack of automation. To investigate the first question, we undertook a study of PSP data quality which found that manual collection and analysis could result in data quality problems that could effect the outcomes and interpretation of the data (Johnson and Disney, 1999, 1998). To investigate the second question, we implemented extensive tool support for PSP/TSP style of data collection and analysis (Johnson et al., 2000), but still found the overhead to be substantial (Johnson, 2001).

Hackystat. In 2001, we initiated a research project called Hackystat (Johnson, 2007; Johnson et al., 2005, 2003), one of whose goals is to support abstract and automated empirical thinking in the classroom setting in a manner different from the PSP/TSP. To accomplish this, we changed the types of data collected and the nature of the analyses and interpretations provided by the

framework. For example, in the PSP/TSP (as well as other approaches like COCOMO), data on completed projects is used to make predictions about future, as-yet-unstarted projects. Hackystat instead uses "sensors" attached to development tools to automatically and unobtrusively collect fine-grained process and product data. Analyses on this data are intended for direct feedback into the current system under development, not for use in future system planning.

For the past five years, Hackystat has been an integral part of the University of Hawaii software engineering curriculum at both the undergraduate and graduate levels. We performed case study experiments in 2003 (Johnson et al., 2004; Johnson, 2003), 2006 (Johnson, 2006), and 2008 (Johnson and Zhang, 2009; Zhang and Johnson, 2009) to assess the classroom impact and effectiveness of the system in supported automated and abstract empirical thinking. Each case study collected both quantitative and qualitative data that motivated extensive redesign and improvement of the system, which we evaluated in the subsequent case study. While the details of evaluation differed in the three studies, in all cases we were generally concerned with three issues: (1) What was the perceived overhead of the system? In other words, how well does the system provide automation? (2) What was the perceived utility of the system? In other words, how well does the system provide abstraction? (3) How well will the system apply to "professional" settings? In other words, to what extent does the empirical thinking promoted by this system feel relevant and useful in the long term?

These case studies generated a great deal of useful data that has directly influenced the course of our research and educational practice. For example, the 2003 experiment provided data indicating that sensor installation was perceived as a significant barrier to use. From the *e*CT perspective, this is an example of a failure of the system to provide sufficient automation. The 2006 experiment provided data indicating that students had difficulties interpreting the trends in data and understanding when the data indicated the need for a change in behavior. From the *e*CT perspective, this is an example of a failure of the system to provide sufficient abstraction. In all three case studies, students have raised concerns about privacy issues. This is an example of a challenge and potential limitation of this approach to support empirical thinking.

Software ICU. In 2008, we performed an initial case study evaluation of a new approach to teaching *e*CT concepts. In this approach, we frame *e*CT within the metaphor of a medical intensive care unit (ICU).

Medical intensive care units feature automatic and continuous monitoring of patient vital signs. The four fundamental medical vital signs are temperature, heart rate, blood pressure and respiration rate. Other vital signs may be monitored depending upon the particulars of a patient condition. Vital signs have a "normal range of behavior", and the monitoring unit can raise an alarm when any of the patient's vital signs departs from its normal range of behavior. Vital signs are interesting because: (a) in a healthy patient, they are normal or improving; (b) change in one vital sign may or may not be significant; (c) change in multiple vital signs is almost certainly significant, particularly if more than one are outside their normal range.

The Software ICU translates "health", "vital signs", "normal range" and the ICU monitoring user interface into terms useful to students and their software development projects. We defined a healthy development project as satisfying three high-level characteristics: high efficiency (software development proceeds "as fast as possible, but no faster"); high effectiveness (effort is focused on the most important issues, with minimal rework); and high quality (software satisfies user needs; software can be easily installed, adapted, and maintained).

We then presented a set of simple practices that, if followed, we claimed would improve

the health of their projects. These included: everyone works consistently; everyone contributes equally; code is committed consistently; progress is regular; quality remains high; no last minute rush to finish. These development practices are analogous to life-style behaviors like "eat right", "get enough sleep" and "exercise regularly" that generally facilitate (but, of course, do not guarantee) good health in a patient.

Next, we presented nine software vital signs: coverage, complexity, coupling, churn, builds, commits, unit tests, size, and development time. Through a combination of Hackystat sensors and the Hudson continuous integration system, these nine vital signs could be automatically and continuously collected for their projects. For each software vital sign, we then presented its normal range of behavior. For example, for the coverage vital sign to be considered healthy, its current value should be above 90% and the trend in coverage over time should be stable or increasing. For the commit vital sign to be considered normal, at least 50% of the team members should have committed, and there should be commits on at least 50% of the days in the project interval. For one of the vital signs, size, we stated that there is no simple way of assessing its normal range of behavior, though it still provides some value in understanding project health.

Unlike a medical ICU, where there is literally hundreds of years of medical research establishing both the importance of the four fundamental vital signs and their normal range of behaviors, no comparable body of knowledge and practice exists for software development. Learning how to assess the usefulness of the selected software vital signs and appropriateness of what we declared as " normal" was an essential part of this *e*CT curriculum.

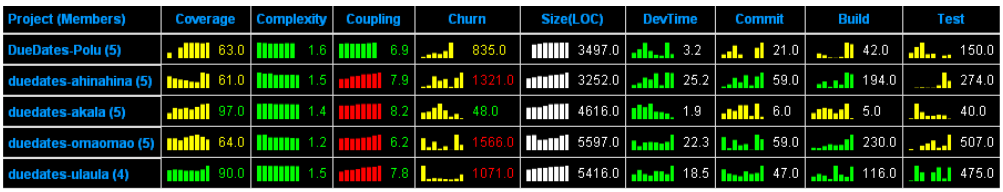

Figure 1: An example Software ICU display

Finally, we presented the user interface to the Software ICU. A portion of this user interface appears in Figure 1. Each row in the Software ICU interface provides information about one software project. Each column presents information about one vital sign. Similar to the medical ICU, the software ICU presents both the most recent numeric value as well as the recent trend in values for each vital sign. The normal range of behavior is represented by independently coloring the trend line and the most recent value as green, yellow, or red depending upon whether the value was healthy, unstable, or unhealthy.

The measurements underlying the Software ICU were collected automatically through two mechanisms. First, the students installed Hackystat sensors into their IDE (Eclipse) and build system (Ant) which sent process metrics regarding their development activities. Second, their projects used the Hudson system to perform continuous integration, which meant that after each commit of their code, the system would be automatically built and tested. The Hudson system was also configured to automatically gather certain product metrics such as coverage, coupling, and complexity.

The results of our initial case study of the Software ICU indicate that it creates a wealth of opportunities for exploring *e*CT concepts in the classroom setting. Students gained awareness of the

strengths and weaknesses of these nine empirical models of software development processes and products. They could observe their own project's behavior over time, compare it to other projects, and see how changes in their development behaviors affected the vital signs. The case study also indicates a number of promising ways to improve the system, which we are implementing in preparation for its next deployment in Fall, 2009.

Devcathlon. A current active research and development project involves the creation of an environment in which *e*CT principles are embedded within a game environment. Unlike other software development games which rely on simulation of developer activities (Drappa and Ludewig, 2000; Navarro and van der Hoek, 2009; Jain and Boehm, 2006), Devcathlon is designed around the use of actual data collected from students as they develop software. Students can form teams and play matches against each other. Matches are based upon "events" which reward teams for appropriate software development behaviors, such as "commit early, commit often", "keep the coverage high", and "don't break the build".

Devcathlon is designed to contrast in interesting ways with the Software ICU. Unlike the passive, "pull-based" interface in the Software ICU, Devcathlon will provide a more active, "push-based" interface in which point awards will can be broadcast to participants via Twitter, email, or instant messaging. The observations are also more fine-grained. For example, in the Software ICU, commit events are aggregated for an entire team and day. In Devcathlon, a single build event can generate a point award (if, for example, the build fails and the match is configured to award negative points for that behavior). Unlike the Software ICU, which can allow every project to be "healthy", Devcathlon requires some projects to "win" and some to "lose". We are interested to evaluate the impact of introducing competition in this way.

As of Spring, 2009, Devcathlon is under active development and we expect to have an initial release for evaluation by Fall of 2009.

4 Implementation Plan

Our implementation plan focuses primarily on three stakeholder groups: university computer science students; their teachers, and computer science pedagogy researchers. It includes the functional activities discussed below, with additional details in the supplementary document called Project Timeline and Milestones.

Common *e*CT Evaluation Framework development. A primary objective of this project is to develop a framework for evaluation of *e*CT initiatives. The goal of this evaluation framework is to elicit useful information concerning the ways in which a particular approach to *e*CT provides for empirical, automated, and abstract thinking. Figure 2 provides an overview of the framework components.

The Framework elicits information regarding six key aspects of an *e*CT initiative: student demographics, curriculum integration, empiricism, abstraction, automation, learning objectives, and outcome data. The structure of this framework, and the questions we pursue within each area, are based upon our prior *e*CT experiences starting with the PSP and up to our current evaluation of the Software ICU. We also incorporate findings from prior software development assessment efforts, such as ATSE (Klappholz et al., 2003).

Addressing the questions in the Framework provides a good basis for understanding the design trade-offs inherent in any *e*CT effort. For example, the PSP sacrifices some potential forms of

Component	Description
Student Demographics	What are the required or desirable characteristics of the student population that appear to make them suitable to this form of *e*CT? What kinds of prerequisite skills or technical background does this form of *e*CT presuppose?
Curriculum Integration	Which course or courses are best suited to this form of *e*CT: introductory, intermediate, or advanced computer programming? Is the *e*CT experience provided as a stand-alone course, a "mixin" occurring throughout the course, or a short, self-contained "module"?
Empiricism	What types of observations are made? Are they qualitative, quantitative, or both? When are the observations made? What are the potential sources of error? Can observations be triangulated and/or cross-validated? What is the potential for measurement dysfunction? What is the overhead on students and teachers?
Abstraction	Into what representations are the observations abstracted? When are these abstractions made? Is abstraction generation student-controlled or teacher-controlled? How are the results of abstraction communicated to students? Is this communication "pull-based", "push-based", or some combination? What is the overhead on students and teachers?
Automation	What forms of automation are used, if any, to: (a) collect observations; (b) generate abstractions from the observations; (c) visualize abstractions; (d) disseminate abstractions; (e) validate abstractions? What kinds of failure are possible? What is the overhead? What are the technical and infrastructure prerequisites?
Learning objectives	What are the intended learning objectives? What knowledge will students have; what skills will they have assimilated; and what attitudes be fostered? How are these learning objectives measured?
Outcome data	What qualitative and quantitative data can be made public? What kinds of contextual information can be provided to support meta-analysis and data mining, without compromising privacy and confidentiality?

Figure 2: Evaluation Framework Components

automated data collection (i.e. time and defects) in order to support certain kinds of abstraction (effort and quality estimation models). Retina sacrifices many kinds of empirical observations in order to address the limited programming capabilities of novice programmers.

The Framework also illuminates opportunities for synergy between initiatives and/or adaptation of innovations from one approach to another. For example, PSP abstractions are mainly focused on project planning improvements through a historical database of past project data. The PSP provides empirical techniques that enable you, based on your prior performance, to make predictions about future project end dates and required resources. However, let's say that you have a team with suboptimal behaviors. In some sense, the PSP can even codify these behaviors, as the team might get simply get better at predicting their suboptimal performance. The Software ICU, on the other hand, focuses purely on improving behavior without providing insight into the "goal line". Combining the two has the potential to address the weaknesses in both.

The development of the Common *e*CT Evaluation Framework will be ongoing throughout the

project. During the first year of the project, framework-related activities will consist of simply collecting information about the ways in which currently known *e*CT initiatives (such as PSP/TSP, Software ICU, Devcathlon, SimSE, and Retina) address these issues. We expect to refine the types of questions we ask and the way we capture the data as part of the first year's activity. The result of this initial phase of data gathering will form our "baseline".

In subsequent years, we will use this baseline data to help push the *e*CT community forward along two dimensions. First, the baseline should help us establish more consistent, higher quality evaluation mechanisms. For example, if one group has developed a particularly good instrument for assessing student opinion, then the baseline can make this apparent and help spread its use to other organizations. Second, the baseline can help assess attempts at synergy. For example, it could help understand what new problems might arise from a composite PSP/TSP/Software ICU approach. It can reveal new opportunities, such as the possibility of adapting the Retina recommendation for use in advanced computer programming classes. Finally, it help reveal opportunities for transfer of insight. For example, both SimSE and the PSP/TSP have been evaluated in multiple university settings, while the Software ICU has not.

On the other hand, it is not our goal for the framework to enable "apples to apples" comparisons, such as "Students using PSP/TSP learn more than students using Devcathlon", or "The Software ICU provides better abstractions than Retina". We believe that the context, demographics, and goals of the current *e*CT initiatives are much too diverse those kinds of comparisons to be valid or have value.

Canonical Learning Objectives Development. According to Mager (1962), learning objectives should include three components: (a) a specific, observable *behavior*; (b) the *conditions* under which the behavior is to be completed, including any tools or assistance; and (c) the *standard* of performance, including any acceptable range of outcomes.

While we do not believe that there can exist a single set of learning objectives that will universally apply to all possible *e*CT initiatives, we do believe that it is possible to create a basic set that can be used as a basis for enhancement or customization. Figure 3 illustrates a preliminary set of canonical learning objectives to be evaluated, refined, and expanded upon in this project.

UH *e*CT curriculum development. Another functional area involves enhancement of our own *e*CT initiatives involving the Software ICU and Devcathlon. We plan to use and evaluate both of these approaches in the software engineering curriculum at the University of Hawaii each year over the course of the project. Feedback from our initial case study (Johnson and Zhang, 2009; Zhang and Johnson, 2009) has surfaced a variety of opportunities for improvement in the Software ICU, and we have yet to deploy Devcathlon in a classroom setting.

We will also use the UH software engineering curriculum as a way to exercise, evaluate, and refine the Common *e*CT Framework and Canonical *e*CT Learning Objectives discussed above, as well as the *e*CT curriculum and outcome data repositories discussed below.

While our prior experience provides a rich set of enhancements to these systems, we look forward to the results of the first year of the project, when the baseline data from the *e*CT Common Evaluation Framework becomes available. This will generate a second source of improvement opportunities for both the Software ICU and Devcathlon, based upon analysis of the strengths, weaknesses, and application of other *e*CT initiatives.

***e*CT curriculum repository development.** To make *e*CT initiatives replicable across institutions, it is necessary to "package" the curriculum, associated technologies for empirical data gathering, abstraction, and automation, as well as evaluation mechanisms.

*e*CT Learning Objective	Description
Awareness	The student can specify six examples of observable behaviors of software products and processes. For each observable behavior, they can specify at least one abstraction that can be generated from that observation, and at least one tool that can support automation in either collection, abstraction, or presentation.
Application	Given a specific software development context, the student can specify at least three observable behaviors of software products and processes. For each of these observable behaviors in this context, the student can indicate a useful abstraction as well as feasible tool support that supports automation in either collection, abstraction, or presentation.
Limitation	Given a specific software development context and a single observable behavior along with an associated abstraction and automation, the student can explain the limitation(s) of this single perspective on software development.
Dysfunction	An observation, along with its abstraction and/or automation, can sometimes be performed in such a way as to misrepresent the actual software development project of interest. For a given type of observation along with its associated abstraction and automation, the student can explain the potential way(s) in which misrepresentation could occur, as well as the reasons why a developer might be motivated to do so.
Single impact	Given a software development context and a single type of observation with its associated abstraction(s) and automation(s), the student can describe what it suggests should change about the way in which software development is done.
Multiple impact	The student can assess how two or more observations with their associated abstractions and automations impact on a given software development context. Specifically, they can state whether the data supports a specific type of change, whether the data is in conflict regarding a specific type of change, or whether the data is not relevant to a specific type of change.

Figure 3: Canonical Learning Objectives

There are a variety of possible packaging mechanisms. For technology, the natural approach is to use open source licensing and one of the several available open source hosting services such as SourceForge or Google Project Hosting. Fortunately, most *e*CT technologies are already open source and employ an open source hosting service. For curriculum materials, we will take advantage of a packaging mechanism such as Open Seminar.

***e*CT public outcome data repository development.** In recent years, there have been significant advances in the availability and sophistication of public repositories for data sets, including CKAN, FreeBase, Swivel, DataPlace, LinkingOpenData, SWSE, and TheInfo. We have participated in collaborative research on appropriate protocols for the use of software engineering scientific data sets and the licensing issues that can result (Basili et al., 2007). We believe that the creation of an online repository of *e*CT outcome datasets, when combined with the contextual

information provided by the Common *e*CT Evaluation Framework, can result in an unparalleled source of useful evidence regarding the state and progress of *e*CT.

For example, the Software ICU gathers a variety of abstractions on a daily basis regarding student projects, including coupling, complexity, coverage, builds, unit test invocations, size, and so forth. This data could be anonymized and uploaded to an *e*CT repository along with general demographic information that would enable teachers at other universities to compare their use of the Software ICU with ours. Other technologies, such as the PSP/TSP or SimSE, could benefit in this way as well.

***e*CT post-course impact assessment.** An important open question for *e*CT initiatives is their long-term impact on the students. A year or more after the course has ended, do they still view their *e*CT experience as useful? If the experience involved tools, do they still use these tools? Have they implemented or adopted other tools that share an *e*CT orientation? Have they grown in their sophistication regarding the appropriate use of empirical computational thinking?

This is a very difficult question to investigate, due to the long time frame involved and the practical difficulties in establishing and maintaining contact with the students. Fortunately, the rise in social networking technologies, in particular Facebook and LinkedIn, provide an approach to this problem. As part of our initial year activities, we will implement a number of Facebook-based mechanisms for creating a community of *e*CT "graduates", including groups, recreational quizzes, and so forth. We will ask instructors of *e*CT courses to inform their students about these social network mechanisms and encourage them to join. Once they are members, we can contact them at yearly intervals to obtain their perspectives on *e*CT.

From empirical to scientific and evidence-based computational thinking. The migration patterns of box jellyfish discussed in Section 2 demonstrates that empirical thinking can be both accurate and useful in the modern, natural world. Similar kinds of associative thinking occur in the artificial world of software development, such as the observation that low levels of test coverage are strongly associated with a system that is difficult to modify and failure-prone.

An important direction for this research is to build upon the skills and techniques for *e*CT to provide students with insight into scientific, evidence-based thinking. We suspect that direct experience of scientific, evidence-based thinking might be impractical within the confines of an undergraduate programming curriculum, as it requires awareness of and the ability to manipulate concepts such as dependent and independent variables and internal and external validity while simultaneously engaged in learning about software development. Prior efforts to incorporate scientific thinking into the software development classroom use simulation rather than direct experience (Höst, 2002).

Rather than require students to incorporate scientific experimentation into their software development process, we propose to use *e*CT as a springboard for discussion of the differences between empirical and scientific thinking, and the costs and benefits associated with the latter. Continuing our previous example, there is a clear association between low system quality and low test coverage that students can experience directly through *e*CT techniques. However, the association between high system quality and high test coverage is much more subtle: while a high quality system tends to strongly associated with high test coverage, the converse is not necessarily true. It is unfortunately all too possible to produce a test suite that exhibits high coverage without actually detecting or preventing important classes of defects (Marick, 1999).

This asymmetry in the association between system quality and test coverage provides an example of the limitations of purely empirical thinking in software development, and can provide

students with an opportunity to discuss ways in which the scientific method can be applied to gain deeper insight into the true relationship between test coverage and system quality.

We will also explore whether the repository of outcome data produced by this project can be adapted to support evidence-based analysis. For example, if an *e*CT student observes that cyclomatic complexity above 25 appears to be associated with classes requiring redesign, they might be able to query the repository in addition to a literature review to gather evidence to support or refute this empirical observation.

***e*CT everywhere.** Although the primary focus this project is to apply *e*CT to upper-level computer science programming courses, empirical computational thinking can be applied elsewhere in the computer science curriculum, as shown by the Retina project. It can even transcend disciplines: for example, one could apply *e*CT to English composition courses by automating the observation of grammatical errors in writing exercises, abstracting these into improvement opportunities, and generating personalized or course-wide recommendations. Our project will establish an experience base that will facilitate the spread of *e*CT throughout the computer science curriculum as well as to other disciplines.

5 Collaboration and Management Plan

The principal leadership of this project will be the responsibility of the PI, Philip Johnson, and his research group, the Collaborative Software Development Laboratory at the University of Hawaii. With over ten years of prior work on curriculum and technology development related to *e*CT, we have demonstrated a long term interest, commitment, and successful track record for this form of pedagogy.

The Supplemental Documents section of this proposal contains letters of support from Pekka Abrahamsson, Teresa Baldassarre, Jeff Carver, Hakan Erdogmus, Tom Hilburn, Letizia Jaccheri, Ross Jeffery, Maurizio Morisio, Dan Port, Carolyn Seaman, Andre van der Hoek, Laurie Williams, and Claes Wohlin. These letters indicate a broad spectrum of support for the research and educational potential of empirical computational thinking, and an interest in seeing this concept integrated into the computer science curriculum.

One of our management priorities is to lower the barriers to adoption. As noted above, we will establish and maintain repositories for technologies, curriculum materials, and outcome data. This helps prospective participants to select the most appropriate *e*CT technology for their needs. Second, we will avoid introducing new overhead on participants by utilizing existing conferences, workshops, and gatherings. Finally, we will leverage existing social network technologies, such as Facebook and LinkedIn, and create *e*CT groups or discussion forums.

During the first year, we will bootstrap collaborations by promoting *e*CT in four venues: ISERN 2009 (October, 2009), CSEET (February, 2010), ICSE (May, 2010), and the TSP Symposium (Fall, 2010). Depending upon the venue, bootstrapping activities could consist of: an informal "birds of a feather" sessions; a poster session; a short talk; or a tutorial session.

By the third year, we believe that there will be sufficient initial adoption and evaluation of *e*CT initiatives to support a workshop for presentation and comparison of experiences, co-located with another conference such as ICSE. At that time we will also work with editors of journals such as IEEE Software, or Empirical Software engineering to propose a special issue on *e*CT.

Our hope is that after these first three years, we will have generated and institutionalized *e*CT such that future engagement and pursuit of this approach is self-sustaining.

6 Evaluation Plan

Our evaluation plan has three components. The first component provides for evaluation of a single *e*CT initiative. To accomplish this, we will use the Common *e*CT Evaluation Framework as described in Section 4. The outcome data sets will also provide evaluation information about a single initiative.

There is a bias in the scientific community against the reporting of negative results. We believe that the long-term success of empirical computational thinking requires a community in which both failure and success are acceptable and seen as equal sources of insight. Thus, our evaluation plan will encourage and support the reporting of experiences regardless of the apparent "success" of the initiative.

The second component of our evaluation plan involves the project as a whole. For this component, we want to assess how well we have been able to create a community of research and practice around the idea of *e*CT. Figure 4 presents some of the metrics we plan to gather on a yearly basis.

Metric	**Description**
Institutions	Number of participating institutions.
Data sets	Number of outcome data sets
Initiatives	Number of distinct, participating *e*CT initiatives
Cross-fertilization	Number of attempts to integrate multiple *e*CT initiatives
Adoption	Number of instances of documented post-course *e*CT use.
Tailoring	Number of instances in which an *e*CT initiative was tailored for use by a new institution.
Publications	Number of publications related to or referencing an *e*CT initiative.

Figure 4: Project-wide Evaluation Metrics

We will gather these metrics at the end of each year of the project. Evidence of the success of this project should be seen by increasing values for most or all of these metrics over the course of the project. By the end of three years, we expect *e*CT participation from at least six institutions; at least ten publicly available data sets; at least two attempts at cross-fertilization; at least three attempts to tailor; and at least six publications in refereed journals or conferences. The Project Timeline and Milestones document in the Supplementary Documents section provides more information on when evaluations of the various functional activities will occur.

The third component of our evaluation plan involves assessing the longer-term impact of *e*CT on students after they complete the course. To do this, we will create an *e*CT group in two social networking technologies: LinkedIn and Facebook. In all participating *e*CT courses, we will ask the instructor to tell students about these groups. Having established contact and community in this way, we will perform follow-up assessments to obtain their perspectives on the long-term merits of their *e*CT experiences. While this will not produce a statistically sound representation, it should still yield insight into post-course use of *e*CT.

7 Anticipated outcomes, intellectual merit, and broader impact

The intellectual merit and broader impact of this project can be summarized in terms of the following anticipated contributions.

First and foremost, this project will create and institutionalize the notion of empirical computational thinking as a useful component for advanced programming courses. Students will learn to observe their programming behaviors and the effect of these behaviors on the system, create and manipulate abstractions of these behaviors, and use automation to increase scalability, precision, and utility. This automation and abstraction is also essential to enabling the continued use of *e*CT concepts as these students move into professional environments and encounter more complex development situations.

Second, this project will create a new community of research and practice around the unifying concept of empirical computational thinking. Currently, this community is fragmented and opportunities for collaboration and synergy are limited. We will create and maintain this community primarily through electronic infrastructure, such as community repositories for technology, curriculum materials, and data. It will also be maintained through joint research and teaching activities. To promote this community, we will perform outreach at related conferences and meetings, and propose workshops at conferences such as ICSE.

Third, this project will generate two new mechanisms for evaluating initiatives in empirical computational thinking: the Common *e*CT Evaluation Framework and the Canonical *e*CT Learning Objectives. These will provide a way to understand, compare, and integrate curriculum initiatives in empirical computational thinking.

Fourth, this project will lead to significantly increased use of *e*CT initiatives in the computer science curriculum. This will occur because this project will lower the barrier to entry for teachers interested in these techniques, who will have a way to more easily determine the *e*CT initiative appropriate to their situation, access to tailorable curriculum materials, and data regarding prior use, outcomes, and challenges.

Fifth, this project will generate new empirical data sets regarding software development activities in a classroom setting. Along with the contextual information provided by the Evaluation Framework, this will create new opportunities for meta-analysis and data mining.

Sixth, this project will serve underrepresented populations, as the University of Hawaii is an EPSCOR state. Approximately 84% of undergraduates at the University of Hawaii are minorities, and the computer science students exemplify this diversity. The software engineering curriculum at the University of Hawaii is well-regarded within the local high tech community, and many of its graduates have gone on to leadership positions. A successful *e*CT initiative could thus be transformative beyond the college and into the local community.

Seventh, this project supports the NSF goal of fostering integration of research and education. The research outcomes regarding *e*CT will impact directly on classroom practice.

Eighth, this project supports the use of *e*CT as a foundation for scientific and evidence-based thinking. It will create an experience base that will facilitate the spread of *e*CT throughout the computer science curriculum and into other disciplines as well.

References

Abrahamsson, P. and K. Kautz, 2002: Personal software process: Classroom experiences from Finland. In *Proceedings of the 7th International Conference on Software Quality*, Springer-Verlag, London, UK, ISBN 3-540-43749-5, pp. 175–185.

Basili, V. R., M. V. Zelkowitz, D. Sjoberg, P. M. Johnson, and T. Cowling, 2007: Protocols in the use of empirical software engineering artifacts. *Empirical Software Engineering*, **12**.

Borstler, J., D. Carrington, G. Hislop, S. Lisack, K. Olson, and L. Williams, 2002: Teaching PSP: Challenges and lessons learned. *IEEE Software*, **19(5)**.

Carrington, D., B. McEniery, and D. Johnston, 2001: PSP in the large class. In *Proceedings of the 14th Conference on Software Engineering Education and Training*, pp. 81–88.

Ceberio-Verghese, A., 1996: Personal Software Process: A user's perspective. In Mead, N. R., ed., *Ninth Conference on Software Engineering Education*, IEEE Computer Society Press, 10662 Los Vaqueros Circle, P. O. Box 3014, Los Alamitos, CA 90720-1264.

Conn, R., 2004: A reusable, academic-strength, metrics-based software engineering process for capstone courses and projects. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, ACM, New York, NY, USA.

Dewey, J., 1910: *How we think*, D.C. Heath, chap. Empirical and scientific thinking.

Dingsoyr, T., L. Jaccheri, and A. Wang, 1999: Teaching software process improvement through a case study. In *Proceedings of the Conference on Engineering and Computer Education*, Rio de Janeiro, Brazil.

Drappa, A. and J. Ludewig, 2000: Simulation in software engineering training. In *Proceedings of the 22nd International Conference on Software Engineering*, ACM, New York, NY, USA, ISBN 1-58113-206-9, pp. 199–208.

Ferguson, P., W. S. Humphrey, S. Khajenoori, S. Macke, and A. Matvya, 1997: Introducing the Personal Software Process: Three industry cases. *IEEE Computer*, **30(5)**, 24–31.

Hayes, W. and J. W. Over, 1997: The Personal Software Process (PSP): An empirical study of the impact of PSP on individual engineers. Tech. Rep. CMU/SEI-97-TR-001, Software Engineering Institute, Pittsburgh, PA.

Höst, M., 2002: Introducing empirical software engineering methods in education. In *Proceedings of the 15th Conference on Software Engineering Education and Training*, IEEE Computer Society, Washington, DC, USA, p. 170.

Humphrey, W. S., 1995: *A Discipline for Software Engineering*. Addison-Wesley, New York.

Jaccheri, L. and T. Osterlie, 2005: Can we teach empirical software engineering? In *Proceedings of the 11th IEEE International Software Metrics Symposium*, Como, Italy.

Jain, A. and B. Boehm, 2006: SimVBSE: Developing a game for value-based software engineering. In *Proceedings of the Conference on Software Engineering Education and Training*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 103–114.

Johnson, P. M., 2001: You can't even ask them to push a button: Toward ubiquitous, developer-centric, empirical software engineering. In *The NSF Workshop for New Visions for Software Design and Productivity: Research and Applications*, Nashville, TN.

—, 2003: Results from the 2003 classroom evaluation of Hackystat-UH. Tech. Rep. CSDL-03-13, Department of Information and Computer Sciences, University of Hawaii, Honolulu, Hawaii 96822.

—, 2006: Results from the 2006 classroom evaluation of Hackystat-UH. Tech. Rep. CSDL-07-02, Department of Information and Computer Sciences, University of Hawaii, Honolulu, Hawaii 96822.

—, 2007: Requirement and design trade-offs in Hackystat: An in-process software engineering measurement and analysis system. In *Proceedings of the 2007 International Symposium on Empirical Software Engineering and Measurement*, Madrid, Spain.

Johnson, P. M. and A. M. Disney, 1998: The personal software process: A cautionary case study. *IEEE Software*, **15(6)**.

—, 1999: A critical analysis of PSP data quality: Results from a case study. *Journal of Empirical Software Engineering*.

Johnson, P. M., H. Kou, J. M. Agustin, C. Chan, C. A. Moore, J. Miglani, S. Zhen, and W. E. Doane, 2003: Beyond the personal software process: Metrics collection and analysis for the differently disciplined. In *Proceedings of the 2003 International Conference on Software Engineering*, Portland, Oregon.

Johnson, P. M., H. Kou, J. M. Agustin, Q. Zhang, A. Kagawa, and T. Yamashita, 2004: Practical automated process and product metric collection and analysis in a classroom setting: Lessons learned from Hackystat-UH. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, Los Angeles, California.

Johnson, P. M., H. Kou, M. G. Paulding, Q. Zhang, A. Kagawa, and T. Yamashita, 2005: Improving software development management through software project telemetry. *IEEE Software*.

Johnson, P. M., C. A. Moore, J. A. Dane, and R. S. Brewer, 2000: Empirically guided software effort guesstimation. *IEEE Software*, **17(6)**.

Johnson, P. M. and S. Zhang, 2009: We need more coverage, stat! Experience with the software ICU. In *Submitted to the 2009 Conference on Empirical Software Engineering and Measurement*, Orlando, Florida.

Kitchenham, B., 2004: Systematic reviews. In *Proceedings of the 2004 International Symposium on Software Metrics*.

Kitchenham, B., T. Dyba, and M. Jorgensen, 2004: Evidence-based software engineering. In *Proceedings of the 2004 International Conference on Software Engineering*.

Klappholz, D., L. Bernstein, D. Port, and P. Dominic, 2003: Tools for outcomes assessment of education and training in the software development process. In *Proceedings of the Conference on Software Engineering Education and Training*, IEEE Computer Society, Los Alamitos, CA, USA, p. 331.

Lisack, S., 2000: The personal software process in the classroom: student reactions (an experience report). In *Proceedings of the 13th Conference on Software Engineering Education and Training*, pp. 169–175.

Mager, R., 1962: *Preparing objectives for programmed instruction*. Fearon, Belmont, CA.

Maletic, J. I., A. Howald, and A. Marcus, 2001: Incorporating PSP into a traditional software engineering course: An experience report. In *Proceedings of the Fourteenth Conference on Software Engineering Education and Training*, pp. 89–97.

Marick, B., 1999: How to misuse code coverage. In *Proceedings of the 16th Interational Conference on Testing Computer Software*, pp. 16–18.

Murphy, C., G. Kaiser, K. Loveland, and S. Hasan, 2009: Retina: Helping students and instructors based on observed programming activities. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, ACM, New York, NY, USA.

Navarro, E. and A. van der Hoek, 2007: Comprehensive evaluation of an educational software engineering simulation environment. In *Proceedings of the Twentieth Conference on Software Engineering Education and Training*.

—, 2009: Multi-site evaluation of simSE. In *Proceedings of the The 40th ACM Technical Symposium on Computer Science Education*, Chattanooga, TN.

Robillard, P. N., 1998: Measuring team activities in a process-oriented software engineering course. In *Proceedings of the 11th Conference on Software Engineering Education and Training*, IEEE Computer Society, Washington, DC, USA, ISBN 0-8186-8326-0.

Valerdi, R. and R. Madachy, 2007: Impact and contributions of mbase on software engineering graduate courses. *Journal of Systems and Software*, **80(8)**, 1185–1190.

Wing, J., 2006: Computational thinking. *Communications of the ACM*, **49(3)**.

Zhang, S. and P. M. Johnson, 2009: Results from the 2008 classroom evaluation of Hackystat. Tech. Rep. CSDL-09-03, Department of Information and Computer Sciences, University of Hawaii, Honolulu, Hawaii 96822.