Assessment Design Patterns for Computational Thinking Practices in Secondary Computer Science:

A First Look

December 2015

SRI Education™
A DIVISION OF SRI INTERNATIONAL

Authors

Marie Bienkowski, Eric Snow, Daisy Rutstein, Shuchi Grover **SRI International**

Acknowledgments

This work originated in a December 2011 workshop supported by an NSF planning grant (CNS-1132232). Attending the workshop was an interdisciplinary group of experts who gave us initial guidance on modeling the computational thinking domain for the purposes of our assessments. Meeting participants included experts in assessment and evidence-centered design: Irvin Katz, Educational Testing Service; Geneva Haertel and Britte Cheng, SRI International; and Gail Chapman, University of California at Los Angeles/Into the Loop. There were experts in computational thinking, computer science, and innovative ways of delivering content in school and afterschool settings: Maggie Johnson, Google; Alex Repenning, University of Colorado, Boulder; Chris Stephenson, Google (formerly at CSTA); and Melissa Koch, Anita Borg Institute (formerly at SRI). The Exploring Computer Science curriculum research and development project was represented by Joanna Goode, University of Oregon; Jane Margolis and Noelle Griffin, UCLA; Jenna Masser, CSTA; and Suyen Moncada-Machado, Los Angeles Unified School District. Experts in program evaluation, especially of information technology- and computer science-oriented curriculum, provided guidance on the use of assessments in research and evaluation: Girlie Delacruz, UCLA/CRESST; Jill Feldman, Westat; and Shaileen Pokress, Project Lead the Way.

After the planning grant and under a new NSF grant (CNS-1240625), in January 2013, early versions of the computational thinking practices patterns were reviewed by an expert panel consisting of Chapman, Goode, Katz, and Stephenson. They were also reviewed during a meeting in June 2013 by that grant's advisory board members, Dan Garcia, UC Berkeley; Jill Denner, ETR Associates; Drew Gitomer, Rutgers University; and Terry Vendlinski, TVATE (formerly at SRI).

We especially acknowledge Irvin Katz and Christian Schunn who conducted two rounds of review on the computational thinking practices design patterns (as well as those for collaboration and communication) in June 2015.

This material is based upon work supported by the National Science Foundation under Grant Nos. CNS-1132232, CNS-1240625, and DRL-1418149. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or any collaborator or partner named herein.

Suggested Citation

Bienkowski, M., Snow, E., Rutstein, D. W., & Grover, S. (2015). *Assessment design patterns for computational thinking practices in secondary computer science: A first look* (SRI technical report). Menlo Park, CA: SRI International. Retrieved from http://pact.sri.com/resources.html

Contents

Introduction

Computer science is a relatively young discipline, yet the effects of its application to design and inquiry reach far into the everyday lives of people. Computer science is a body of knowledge that includes new *terminology* related to computer software and hardware operations (e.g., pipelining, caching, CPU, and recursion), and new *languages* to express computational notions. Although opinions differ on whether students need to develop expertise in programming alone (Grover 2013; Resnick, 2013), agreement is growing (Gallup, 2015; Berdik, 2015; Shein, 2014; della Cava, 2015) that students should be exposed to the core ideas and ways of thinking that constitute computer science—computational thinking—in part because such knowledge can transform how science and engineering are practiced (Nature, 2006). For example, computational biologists use algorithms to model phenomena as complex as evolutionary pressures on biological systems and massive data problems such as gene sequencing. Not only does computational thinking help scientists model systems and explore large datasets, it also presents them with new ways of explaining phenomena using computational metaphors (Regav & Shapiro, 2005) and engenders new interdisciplinary communities to study science (e.g., Tadmor & Tidor, 2005). Indeed, computers are now "essential components of modern biological research, and scientists are being asked to adopt new skills in computational biology and master new terminology" (Loman & Watson, 2013, p. 996).

Computer science education is expanding in K-12 education, and research is showing links between learning computational thinking and learning science, technology, engineering, and mathematics (STEM) content (e.g., Basawapatna et al., 2011; Basu et al., 2013; Grover, Pea, & Cooper (2015); Lewis & Shah, 2012; Wilensky & Reisman, 2006). For example, Sengupta and colleagues (2013) used agent-based modeling and computational thinking to support learning in physics and biology and found gains in students' reasoning using mathematical representations and causal reasoning in ecology. Consistent with the modeling practices of computational scientists, researchers have studied how students can use tools such as NetLogo or AgentSheets to build models or simulations of scientific phenomena (Basawapatna, Koh, Repenning, & Lewis, 2013; Wilensky & Reisman, 2006). The projects that use computational thinking in STEM learning support K-12 students' achievement of performance expectations that call for modeling as a way to understand science (NGSS, 2013).

To teach computational thinking effectively, K-12 educators must understand what students know and how learning progresses. Assessing learners' knowledge of new definitions and programming language commands is simpler than discerning how they use computation and programming languages to solve problems and design creative computational artifacts. Assessing how learners apply their knowledge to think computationally means searching for evidence of deeper understanding of the connection between problems to solve and the comprehension and production of coded solutions. How, then, can we best measure how students think about problem solving with computation, apply abstraction, understand others' computational work, and design and implement creative solutions to problems?

This report gives an overview of a principled approach to designing assessment tasks that can generate valid evidence of students' abilities to think computationally. *Principled assessment* means designing assessment tasks to measure important knowledge and practices by specifying chains of evidence that can be traced from what students do (observable behaviors) to claims about what they know. Our approach to assessment produces documents called *design patterns* that provide an overview of how tasks can be designed, and a template for designing them, to elicit evidence about a students' ability in constructs of interest. Design patterns are meant to be generative of multiple tasks and to guide assessment specialists in developing tasks to assess both knowledge and skills in the context of specific learning experiences.

The design patterns presented in this report were developed under the project Principled Assessment of Computational Thinking (PACT). In that project, we worked closely with a particular high school computer science curriculum that has a strong focus on inquiry teaching and equity in access to computing called Exploring Computer Science (ECS). Our long-term objectives in the ongoing PACT suite of projects (under NSF awards CNS-1132232, CNS-1240625, CNS-1433065, and DRL-1418149) are as follows:

- Analyze and model the computational thinking domain to elicit its underlying knowledge and skills and develop artifacts— standards mappings, design patterns, and tasks—to structure the assessment design and development for the knowledge and skills.
- Develop and validate measures of computational thinking by instantiating design patterns in the context of specific curricula to guide assessment design up to and including implementation, with an emphasis on delivering assessments online.
- Identify the implementation factors for ECS and other computer science-related curricula that influence secondary students' learning of computational thinking through the use of assessments and other measures.

To reach these objectives, we have pursued specific short-term goals:

- Create design patterns for major computational thinking practices and ECS units that can be used to design and then refine assessments as curriculum evolves.
- Develop templates for assessment task development for computational thinking practices in the context of ECS.
- Create, pilot-test, and validate for ECS four unit assessments (ECS Units 1-4) and a cumulative assessment (across Units 1-4), including the delivery of the assessments online.

This report addresses our first short-term goal by presenting a general set of design patterns that cut across different curricula for computational thinking. This set of design patterns models a subset of computer science knowledge: that portion that concerns computational thinking practices, or the **application** of design and inquiry to solving computational problems and creating computational artifacts. This set is general to practices in the computational thinking domain and emphasizes the application of design and inquiry skills to solve computational problems (rather than just the underlying conceptual knowledge needed to apply such skills).

Development of the design patterns began in 2011 through interactions with various experts and working groups in computer science education and assessment. At the same time, the community of educators developing Exploring Computer Science

and the new Advanced Placement (AP) Computer Science Principles codified their broad definition of computational thinking *practices* (Arpaci-Dusseau et al., 2013). Adding "practices" to computational thinking reflects an orientation toward not just internal individual "thinking" but "ways of being and doing" that students should exhibit. This orientation is consistent with the Next Generation Science Standards (NGSS) (NGSS Lead States, 2013), which reflect the conviction that science *practices* need to be included in any curriculum or assessment of deeper kinds of knowledge and skill. We aligned our development work with this orientation not only because of the central position of computational thinking practices in secondary computer science education, but also because *practices* are difficult to assess and our experience in principled assessment design positioned us to successfully address this critical need.

This report presents four computational thinking practices design patterns and two supporting design patterns (Exhibit 1) that can guide the development of assessments at a secondary school level. The supporting constructs, collaboration and communication, are ways of enacting practices that cut across specific disciplines. As of this writing, these six design patterns have not yet been used for assessment item development. To help readers understand their utility, for each computational thinking design pattern, we illustrate how it could be applied in ongoing teaching projects: computational thinking in science in the Computational Thinking in STEM (CT-STEM) project (ct-stem.northwestern.edu), constructionist pedagogies using Scratch (scratch.mit.edu), game and simulation design using AgentSheets (www.agentsheets.com), and storytelling and game design using Alice (www.alice.org).

In several new NSF-funded projects, SRI is using these design patterns as a starting point for thinking about assessment in science and math and in other learning contexts outside formal computer science courses. We welcome input from the computer science education and assessment communities about the applicability of these design patterns to assessment development in other computer science education contexts and related content areas.

This report presents an overview of the computational thinking domain, summarizes the approach used to develop the design patterns, and presents the main portions of the design patterns for computational thinking practices with accompanying illustrative applications and the design patterns for collaboration and communications.

Exhibit 1: Computational Thinking Practices and Supporting Constructs

Analyze the effects of developments in computing.
Design and implement creative solutions and artifacts.
Design and apply abstractions and models
Analyze their computational work and the work of others.
Communicate thought processes and results
Collaborate with peers on computing activities

Computational Thinking

Wing (2006, 2011) defined *computational thinking* as the thought processes involved in formulating problems and their solutions so that the solutions are represented in a form that can be effectively carried out by a computer. Computational thinking is now recognized as a concept that encompasses the pervasiveness of computer science constructs and problem-solving strategies such as abstraction at different hierarchical levels, algorithmic thinking, automation, decomposition, modeling, patterns, recursion, scale, and symbolic representations (e.g., Cortina, 2007; Denning, 2009; Grover & Pea, 2013; Guzdial, 2008). In 2010, the National Research Council (NRC) convened a group to study the scope and nature of computational thinking, and the resulting report (NRC, 2010) emphasized that thinking computationally is not completely synonymous with computer science, computer and technology literacy, or programming and that it differs from mathematical, scientific, and quantitative thinking. The working group left computational thinking undefined, however, so follow-on work pursued succinct definitions by examining workforce skills, literature synthesis, and curriculum standards.

For professionals in various disciplines, computational thinking skills both complement and extend other 21st century skills such as collaboration and creativity. In a 2012 workforce study, Malyn-Smith and Lee defined a professional with computational thinking skills as one who is able to collaborate and engage "in a creative process to solve problems, design products, automate systems, or improve understanding by defining, modeling, qualifying and refining systems, processes or mechanisms generally through the use of computers" (p. 3).

Computer programming studies, in part, inspired the shift to computational thinking. A recent critical examination of the current state of practice of K-12 computational thinking drew from and synthesized that literature into the following features (Grover & Pea, 2013):

- Abstractions and pattern generalizations (including models and simulations)
- Systematic processing of information
- Symbol systems and representations
- Algorithmic notions of flow of control
- Structured problem decomposition (modularizing)
- Iterative, recursive, and parallel thinking,
- Conditional logic
- Efficiency and performance constraints
- Debugging and systematic error detection.

These are important characteristics that can inform a comprehensive view of computational thinking. For K-12 stakeholders, definitions for computational thinking seem to downplay programming and instead emphasize problem solving and data representations. Professional organizations for computer science and information technology, namely the International Society for Technology in Education (ISTE) and the Computer Science Teachers Association (CSTA), define computational thinking as follows:

> formulating problems in a way that enables us to use a computer and other tools to help solve them; logically organizing and analyzing data; representing data through abstractions such as models and simulations; automating solutions through algorithmic thinking (a series of ordered steps); identifying, analyzing, and implementing possible solutions with the goal of achieving the most efficient and effective combination of steps and resources; and generalizing and transferring this problem solving process to a wide variety of problems. (Barr, Harrison, & Conery, 2011)

Others have not attempted to be so comprehensive but instead have highlighted features that are consistent with their approaches to using computing to solve problems and design artifacts in STEM and other domains. For example, industry-sponsored websites with resources for educators emphasize elements including patterns, abstractions, models, algorithms, and data analysis and visualization ("Exploring Computational Thinking, n.d.) or give examples of computational thinking in other sciences (Phillips, 2009; Microsoft Corporation, 2014). Some curricula for games and simulations emphasize computational patterns that capture phenomena in modeling agent behaviors and interactions such as diffusion, collision, seeking, and polling ("Scalable Game Design Wiki", 2014). Projects with an orientation to computational science tend to emphasize data, modeling, and systems thinking ("Computational Thinking in STEM: Lesson Plans," 2015). Still, others use computational thinking to teach mathematics ("Bootstrap Materials: Curriculum and Software," 2015).

Most examples or definitions suggest that computational thinking requires integration and application of knowledge in the context of the discipline it is being used in. This framing is consistent with expectations outlined in the Common Core State Standards (National Governors Association, 2010) and NGSS (NGSS Lead States, 2013) in which K-12 students are expected to apply content knowledge in practice. The Common Core standards directly reference computational thinking practices in mathematics, such as problem solving (mathematical practice 1: make sense of problems and persevere in solving them) and abstraction (mathematical practice 2: reason abstractly and quantitatively) (National Governors Association, 2010). The NGSS do not define computational thinking but describe "using mathematical and computational thinking" as an essential practice for modeling and analyzing and interpreting data (NGSS Lead States, 2013).

We can draw from the Common Core and NGSS to similarly define frameworks and standards for computational thinking, but definitions (as above) and curriculum standards for computational thinking as a design and inquiry practice are needed for supporting assessment development (Bienkowski, 2015; Sykora, 2014). Fortunately, performance standards related to computational thinking are replacing information technology fluency standards and recalibrating what college-ready students should know. The CSTA and ISTE have established computer science curriculum standards for three levels of knowledge in K-12 based on input from the professional computing community and computer science educators (CSTA, 2011), and these computer science standards include a strand specifically for computational thinking. The 2011 CSTA standards are undergoing revision and will be informed by a framework for computer science in K-12 currently under formulation by a multistakeholder team led by the computer science advocacy organization Code.org.

Examining definitions and standards, as well as learning objectives for specific courses such as ECS and AP Computer Science Principles, is an important starting point for developing assessment design materials for computational thinking. But standards and curriculum learning objectives are insufficient specifications for developing assessments in that they do not provide the level of content and measurement detail required for the design of assessment items and tasks. A complementary approach is thus required, as described next.

Evidence-Centered Design

To develop assessments for computational thinking practices, we need to define the practices at a level of specificity that makes them amenable to valid and reliable measurement. SRI Education specializes in using the evidence-centered design (ECD) framework to develop assessments for hard-to-assess constructs. ECD is especially helpful when the knowledge and skills to be measured involve complex, multistep performances, such as those required in computational thinking. ECD helps us refine broad learning goals organized around constructs, describe the kinds of tasks and situations that would elicit evidence of a student's ability, and describe how the evidence can be aggregated to produce information on a student's ability.[1] Our experience in these difficult-to-assess domains has been that the up-front design work ECD requires lays the groundwork for efficiently generating families of assessment items that have content validity.

ECD makes explicit, and provides tools for, the building of assessment arguments (Mislevy & Riconscente, 2006; Mislevy, Steinberg, & Almond, 2003). Messick (1994) summarized the essence of the assessment argument as follows.

> A construct-centered approach would begin by asking what complex of knowledge, skills, or other attributes should be assessed, presumably because they are tied to explicit or implicit objectives of instruction or are otherwise valued by society. Next, what behaviors or performances should reveal those constructs, and what tasks or situations should elicit those behaviors? Thus, the nature of the construct guides the selection or construction of relevant tasks as well as the rational development of construct-based scoring criteria and rubrics. (p. 17)

ECD supports developers in building assessment arguments in part by fostering more critical understanding of a domain like computational thinking through systematic generation of design documents—design patterns and task templates—that provide details on the core competencies (Messick's "complex of knowledge, skills, and other attributes") and how they can be measured. These design documents capture all aspects of work from construct specification down to the necessary technical specifications to develop and deliver assessments. Design patterns represent a structured, narrative description of the evidence argument for a domain in way that assessment designers could use them to produce tasks (Liu & Haertel, 2011).

ECD is typically described in terms of five layers of work (Mislevy et al., 2002). These layers and examples of key entities/artifacts created along the way are shown in Exhibit 2. Although the layers suggest steps in a sequential design process, cycles of iteration and refinement are intended, both within and across layers, and work in different layers can occur simultaneously. This report presents work in the domain analysis and modeling layers for the computational thinking domain and suggests how the outcomes could be applied to create assessments of computational thinking practices for both formal and informal education contexts. Domain analysis helped us identify and define the core practices, and domain modeling resulted in design patterns that specified elements for assessment design.

[1] More information on ECD can be found in publications by Mislevy and Riconscente (2005, 2006) and Haertel et al. (in press-b). For examples of domains where ECD has been applied, see DeBarger and Snow (2010) for life sciences; Mislevy, Riconscente, and Rutstein (2009) for model-based reasoning; and Cheng, Ructtinger, Fujii, and Mislevy (2010) for systems thinking.

Exhibit 2. The Five Layers of Evidence-Centered Design and Key Entities for Computational Thinking

ECD Layer	Role	Key Entities & Examples
Domain analysis	Gather substantive information about the computational thinking domain of interest that has implications for assessment; how knowledge is constructed, acquired, used, and communicated	Computational thinking domain concepts (e.g., abstraction, automation); terminology (debugging); tools (programming languages); representations (storyboards); situations of use (modeling predator-prey, visual storytelling), and curriculum standards and mappings
Domain modeling	Express assessment argument in narrative form based on information from domain analysis	Specification of knowledge, skills, and other attributes to be assessed (e.g., describe result of running a program on given data); features of situations that can evoke evidence (find errors in programs); kinds of performances that convey evidence (use of recursion)
Conceptual assessment framework	Express assessment argument in structures and specifications for tasks and tests, evaluation procedures, measurement models	Student, evidence, and task models; student, observable, and task variables; rubrics; measurement models; test assembly specifications; task templates and task specifications
Assessment implementation	Implement assessment, including presentation-ready tasks and calibrated measurement models	Tasks, task materials (including supporting materials, tools, affordances); pilot test data to hone evaluation procedures and fit measurement models
Assessment delivery	Coordinate interactions of students and tasks: task- and test-level scoring; reporting	Tasks as presented; work products as created; scores as evaluated

Source: From Haertel et al. (in press-a).

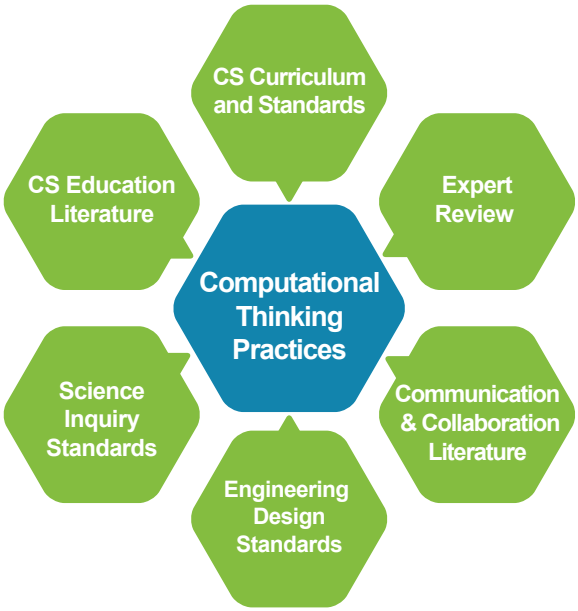
Domain Analysis

The main activity for domain analysis is the review of existing material on the topic of interest (Exhibit 3). For the work here, the sources included construct definitions; standards; curriculum; literature in computer science education, scientific inquiry, engineering design, communication, and collaboration; and previous computer science assessment projects [such as Herman, Loui, & Zilles (2010), McCracken et al. (2001), and Tew & Guzdial (2010) for postsecondary and Werner, Denner, Bliesner, & Rex (2009), Nicholson, Good, & Howland (2009), Robertson & Howells (2008), and Brennan & Resnick (2012) for middle school)].[2]

We also sought input from experienced computer science teachers and received critical feedback from our experts and advisors (see Acknowledgments). We paid special attention to the CSTA standards (CSTA 2011), the ECS curriculum (including learning objectives), and the AP Computer Science Principles framework (including learning objectives and evidence statements). In particular, the ECS and AP Computer Science Principles adoption of four computational thinking practices and two supporting practices served as the basis for the constructs for which we created design patterns.

[2] A source bibliography is at the end of the report.

Exhibit 3. Domain Analysis Sources for Computational Thinking Practices, Communication, and Collaboration



The idea of practices in computer science curricula is consistent with other frameworks. For example, the NGSS Framework (NRC, 2012) specifies that each performance expectation must combine a relevant practice of science or engineering with a core disciplinary idea and crosscutting concept. This is to support assessments that do not measure students' understanding of core ideas separately from their abilities to use the practices of science and engineering. Instead, practices and content will be assessed together, ensuring that students not only know concepts, but that they also can use their understanding to solve meaningful problems through the practices of science inquiry and engineering design. The NGSS Framework uses the term "practices" rather than "science processes" or "inquiry" skills for a specific reason:

> We use the term "practices" instead of a term such as "skills" to emphasize that engaging in scientific investigation [and engineering design] requires not only skill but also knowledge that is specific to each practice. (NRC, 2012, p. 30)

An important design pattern element that emerges from domain analysis and that is refined during domain modeling is the overview statement, a summary of the construct. Overviews for the key domain constructs are used to help generate the design patterns during domain modeling.

The overviews for the computational thinking practices and for communication and collaboration are shown in Exhibit 4. Also shown are analogs, from early work on the AP Computer Science Principles, to these overviews. These analogs indicate how the practices we identified as foundational are consistent with the definitions of computational thinking practices in the Computer Science Principles program.[3] Despite this consistency, we intend for the design patterns to serve as assessment development guides for any curricular program aligned with the computational thinking practices rather than a specific curriculum.

[3] These are from the original definitions of computational thinking practices as described at: https://csprinciples.cs.washington.edu/sixpractices.html

Exhibit 4. Computational Thinking Practices, Overviews and AP Computer Science Principles Analogs

Construct	Design Pattern Overview	CS Principles Analog
Analyze the effects of developments in computing	This design pattern supports the development of assessment tasks in which students show that they understand the range of problems to which computers and computing can be applied. Students will be asked to recognize aspects of computers and computing. They will show an understanding of how computing has enabled innovations in various disciplines and in society as a whole and at the same time has given rise to ethical (e.g., privacy) and social justice (e.g., equal access) issues. They will also demonstrate a broad understanding of "intelligent" machines and the idea of networked systems.	Identification of existing and potential innovations enabled by computational technology Identification of ethical implications of developments in computing Identification of the impacts (positive and negative) of computing innovations on society Analysis of implications of design decisions Evaluation of the usability of a computational artifact Characterization of connections between human needs and computational functionality Explanation of relevant intellectual property issues
Design and implement creative solutions and artifacts	This design pattern supports the development of tasks in which students translate novel ideas and problem solutions into computational solutions and artifacts (i.e., artifacts that involve computing, such as robotics, and/or solutions that are implemented as computer programs). Students design and implement according to a given purpose or intent, including for creative expression. (In this way, programming can be viewed as a communicative act.) This design pattern encompasses steps of both problem-solving and creative processes, including understanding, decomposing, exploring (e.g., by creating different representations of the problem with storyboards, flow charts, pseudocode), creating products that show one or more designed solutions and/or artifacts, and testing and improving the solution and or artifact.	Creation of an artifact chosen by the student as relevant and interesting. Design of a solution to a stated problem. Selection of an appropriate approach to solve a problem Appropriate use of predefined algorithms Appropriate use of programming constructs and data structures Evaluation of an artifact using multiple criteria Location and correction of errors Use of appropriate technique to develop a computational artifact
Design and apply abstractions and models	Thinking strategically about abstraction is a hallmark of computational thinking. This design pattern supports the development of tasks in which students use ideas and representations that capture general-to-specific aspects, or patterns, of an entity or a process and the relationships/structures among entities or processes, including level of detail. This may include designing general solutions to problems or generalizing a specific solution to encompass a broader class of problems (functional abstraction). These ideas and representations may be used in different contexts (problem or disciplines). Students demonstrate knowledge of the representational properties of discrete mathematics, models, diagrams, computer programs (data abstraction), items found in the natural and man-made world, and others. They also demonstrate an understanding of the limitations of models to represent phenomena and an attention to the purpose of the model or abstraction.	Explanation of how data, information, and knowledge are represented for computational use Use of simulation to investigate posed/existing questions and develop new questions Selection of algorithmic principles at an appropriate level of abstraction Use of different levels of abstraction Specification of the design for a model/ simulation Use of data abstractions Collection or generation of data appropriate to a phenomenon being modeled Comparison of generated data to an empirical sample

Exhibit 4. Computational Thinking Practices, Overviews and AP Computer Science Principles Analogs (Continued)

Construct	Design Pattern Overview	CS Principles Analog
Analyze their computational work and the work of others	This design pattern supports the development of tasks in which students demonstrate that they can evaluate computational work (resulting in artifacts such as a program, program outputs, a website, or problem solution) and compare multiple computational artifacts. Students are able to recognize how different techniques can be used to solve problems or achieve computational goals in different ways. Students show they can evaluate the sufficiency (does it meet requirements?), accuracy (does it operate correctly?), efficiency (does it use computing resources wisely?), and elegance (does it follow guidelines of modularity, parsimony, adherence to language-specific and language-general coding idioms, simplicity, well-structuredness, and comprehensibility?) of the artifact. They will demonstrate an understanding of how the forms that computers can take (including robotics) and their user interfaces affect usability (a special criterion for user-facing artifacts).	Identification of problems and artifacts that have a given property Comparison of tools available to solve a problem Evaluation of a proposed solution to a problem and implications of that solution's use Analysis of solution trade-offs with appropriate justification of possible solutions Analysis of the result of a program Evaluation of characteristics of problems and artifacts
Communicate thought processes & results	Communicating about computational artifacts supports many phases of computational thinking. This design pattern supports the development of tasks in which students show that they can communicate the process and results of their work in a way that is appropriate for the particular audience. Students can articulate major themes and ideas related to computing in writing and orally, supported by graphs, visualizations, and computational analysis.	Explanation of the meaning of results Description of the impact of a technology or artifact Summarization of the behavior of a computational artifact Explanation of the design of an artifact Description of technology or artifact Justification of the appropriateness and correctness
Collaborate with peers on computing activities	This design pattern supports the development of tasks in which students demonstrate their ability to engage in the collaborative aspects of computer science by jointly solving problems and designing/creating computational artifacts. Students show they can share understanding and effort, and can pool knowledge and skills with peers, experts, and others using collaborative practices such as pair programming and working in pairs or teams.	Application of effective teamwork practices Collaboration of participants Production of artifacts that depend on active contribution from multiple participants Documentation describing the use, functionality, and implementation of an artifact

Additional design pattern elements specified during domain modeling describe what is within and outside the scope of the construct and give suggestions and guidance for assessment developers. The main elements of design patterns are described next.

Domain Modeling

Domain modeling in ECD produces narrative elements describing a domain for measurement, and the narrative is captured in design patterns. Design patterns in ECD provide specifications for the important ideas to be measured in the assessment and can be used, reused, and refined to help generate many different forms of assessments. For example, one pattern could be used to generate a paper-and-pencil test, an online interactive test, or a rubric to score computational artifacts that students produce. Design patterns are sufficiently general to guide measurement of learning by traditional paper-pencil delivery, as well as by computer-based assessments and even by observations of strategies, tactics, and moves in game play or students' moves in simulation-based and tutored learning environments.

Design Pattern Elements

Design patterns consist of elements specifying all or part of a construct domain in terms of the knowledge and skills to be measured, observations or behaviors that can be used as evidence of knowledge and skills in that domain, and tasks or activities that elicit the desired observations or behaviors. Exhibit 5 provides general definitions of the elements that appear in all the design patterns that we use.

Exhibit 5. Elements of a Design Pattern

Focal knowledge, skills, and other attributes (FKSAs)	The primary KSAs targeted by the design pattern and what we want to make inferences about. For our initial work on computational thinking practices, we focused on skills rather than knowledge.
Additional KSAs	Other KSAs that may be required for successful performance on the assessment tasks but are not the target skills that we are trying to assess. For computer science, this may include knowledge of mathematics or programming languages and tools. Additional KSAs may also be used to link across design patterns to show the interdependencies among skills.
Potential observations	Features of the things students say, do, or make that constitute the evidence on which the inference about a student's performance will be based. Potential observations are described using such qualities as accuracy, degree, completeness, and precision.
Potential work products	Some possible artifacts or observations that one could see. Work products are scored during assessment delivery.
Characteristic features	Aspects of assessment situations that are likely to evoke the desired evidence or that are required to support the task.
Variable features	Aspects of assessment situations that can be varied in order to shift difficulty or emphasis.

Focal Knowledge, Skills, and Attributes

After we have a good overview from the domain analysis, we specify all the focal knowledge, skills, and attributes underlying the construct. Focal here means central or core—the knowledge, skills, and other attributes related to the student that we want to assess. The FKSAs should cover the main ideas within the construct of interest.

The full set of FKSAs for the construct "design and implement creative solutions and artifacts" is presented in the next section. To illustrate the level of detail used in a FKSA, we show five below (Exhibit 6). Note that we deliberately stated our FKSAs as "Ability to…" in order to capture the focus on practices, which we believe are best represented as the application of skills. This

is in contrast to FKSAs that may be better captured as knowledge statements. Complete modeling of a domain requires both, however, and we are currently expanding our design patterns to include knowledge-focused FKSAs to capture the computer science conceptual knowledge underlying the practices.

The abilities we describe in our FKSAs are practices that students should be learning. We attempted to restrict our vocabulary for these abilities—the verbs that describe what students should be able to do—to a finite set that we can measure. For example, we have used *name/identify, describe, design/develop/generate, explain, justify, represent, present, publish, summarize, discuss, provide useful feedback, listen* and not vague terms such as *integrate, recognize, understand, or conclude.*

Exhibit 6. Example Focal Knowledge and Skills for the Construct "Design and Implement Creative Solutions and Artifacts"

1. Ability to state a problem in order to identify the inputs and outputs of the problem
2. Ability to decompose a problem into multiple subproblems, including the specification of how solving the subproblems will lead to a solution to the problem as a whole
3. Ability to create a computational artifact given a purpose or intent
4. Ability to select appropriate techniques to develop computational artifacts
5. Ability to identify run-time errors
6. Collaborate with peers on computing activities

Although we did not impose any hierarchy on these abilities (as we are not describing a progression of learning), we did differentiate the degree to which students should be able to apply their knowledge by using different verbs. For instance, for some skills we thought that it was enough for students to be able to *describe* the phenomenon, whereas for other content we wanted students to be able to *explain*. For our purposes, explanation includes drawing relationships, engaging in interpretation, and comparing and analyzing. In other areas we indicate that students should engage in the context at a higher level by stating the FKSAs as the ability to *evaluate*. Evaluate encompasses explain, justify, and compare.

Potential Observations and Work Products

After specifying the FKSAs underlying a construct, we specify what we could observe the student doing or producing and how those behaviors or artifacts could provide evidence of the FKSAs. Potential observations and associated work products are shown in Exhibit 7 for one FKSA from the construct "design and implement creative solutions and artifacts." Note that by "potential" we intend to capture the idea that these elements can and should evolve as knowledge of the constructs evolves based on research and practice.

What we observe students say and do is more than just the work product, but is the work product and its characteristics. For example, we look for the quality—described as accuracy and appropriateness—of the work product (Exhibit 7). We can also look for correctness but must be careful of cases where there is more than one right answer: In those cases we look for the students' process and justification for why they used that process. We also look for appropriateness and the degree of and extent to which a work product exhibits a given characteristic. The potential observations list these characteristics, which are then further specified when the rubrics are developed.

Exhibit 7. Potential Observations and Work Products for the Construct "Design and Implement Creative Solutions and Artifacts"

Example Potential Observations for one FKSA in "Design and Implement Creative Solutions and Artifacts"
1. FKSA: Ability to state a problem in order to identify the inputs and outputs of the problem
1.a. Accuracy of the statement of a problem
1.b. Appropriateness of the inputs and outputs identified
Example Potential Work Products for one FKSA in "Design and Implement Creative Solutions and Artifacts"
1. FKSA: Ability to state a problem in order to identify the inputs and outputs of the problem
1.a. The statement of a problem
1.b. The list of inputs and outputs of a problem

Characteristic and Variable Features

While thinking about what we can observe students doing and producing, it is natural to think about important features of the *tasks* that measure them. It is also natural to think about ways a task can be varied to make it easier or harder, to remove barriers due to language or culture, or other issues. An ECD design pattern captures these as characteristic and variable features.

Characteristic features are features that any task developed from the design pattern should incorporate and that must be present in all tasks that measure the construct of interest. Variable features are features that can vary and may or may not be present in a particular task measuring the construct of interest. How features vary depends on the measurement goals for the task and could involve changing the difficulty of an item or allowing for additional KSAs to be measured or supported. Specifying the variable features ahead of time helps highlight decisions that should be made when developing items. Characteristic features specify the *required* features of a task, features that must be present in order for the task to elicit evidence of the FKSAs.

Design Pattern Development and Evaluation

During domain modeling, our computational thinking practices design patterns underwent several rounds of review that included critiques by the external experts and advisors in computer science education, assessment design for information technology literacy, inquiry assessment, and curriculum development. All reviewers were given curriculum and standards materials as background. For each design pattern, reviewers were asked the following:

- Is the overall description of the computational thinking practice accurate and comprehensive?
- Are the important FKSAs represented?
- Are the potential observations (what we are willing to count as evidence) correct?
- Are the potential work products (what we expect students to be able to do) appropriate?
- Are the FKSAs well matched with the potential work products and potential observations?

After each round of review we further refined the FKSAs and other design pattern elements.

Computational Thinking Practices Design Patterns and Examples

The following four subsections present the design pattern elements for each of the four core computational thinking practices:

1. analyze the effects of developments in computing
2. analyze their computational work and the work of others
3. design and apply abstractions and models
4. design and implement creative solutions and artifacts

For each design pattern, we present an example of an introductory computer science learning experience that aligns with the FKSAs specified. This alignment demonstrates the broad applicability of each pattern and suggests what could be measured in each context. We then present the two design patterns for communication and collaboration.

Analyze the Effects of Developments in Computing

Overview

This design pattern supports the development of tasks in which students show that they understand the range of problems to which computers and computing can be applied. Students will be asked to recognize aspects of computers and computing. They will show an understanding of how computing has enabled innovations in various disciplines and in society as a whole and at the same time has given rise to ethical (e.g., privacy) and social justice (e.g., equal access) issues. They will also demonstrate a broad understanding of "intelligent" machines and the idea of networked systems.

Focal Knowledge, Skills, and Attributes

General Knowledge of Computing and Computers

1. Ability to recognize that computers are devices that execute programs.
2. Ability to describe the differences between how a computer performs a task and how a human performs a task. This includes the idea that computers follow instructions and that the instructions must be precise and not open to interpretation.
3. Ability to describe the characteristics that make an artifact computational.
4. Ability to describe the components of a computer that enable the behaviors characteristic of computers such as a computational processor – related to what makes a computer a computer.

Use of Computers/Computing

5. Ability to describe major changes in the computing and storage capacity of computers as well as changes in networking capability and form factor (e.g., laptop, smartphone, tablet) due to innovations in computing – includes changes in both hardware and software.

6. Ability to describe a change in the use of computers due to innovations in computing – includes both innovations in computer hardware and software.
7. Ability to describe a way in which computing enables innovation, i.e., design and creation of new products, processes, or services that represent an advance in a field or allows people to do/accomplish something in new ways.
    7.a. Ability to describe an example of how computer programs are used to process information to gain insight and knowledge (including the use of computing tools to work with and make sense of large volumes of data in ways that humans cannot).
    7.b. Ability to describe an example of how computer programs can be used to develop new products, processes, or services that represent advances in various fields.
    7.c. Ability to describe an example of how advances in computing can help in dissemination of new products, processes, or services.
    7.d. Ability to describe an example of how advances in computing can increase communication in order to develop new products, processes, or services.

Knowledge of the Internet and the Systems Built On It

8. Ability to describe multiple uses of the Internet and the systems built on the Internet.
9. Ability to explain why hierarchy and redundancy are important in the implementation of the Internet.
10. Ability to describe interfaces and protocols that enable widespread use of the Internet and systems built on the Internet.
11. Ability to explain how working in networked systems impacts privacy (such as personal data, actions, location, etc.).

Computing in Everyday Life

12. Ability to recognize when computing or computational solutions are applied to other (noncomputer science or information technology) disciplines.
13. Ability to describe how computers and computing aids are used in diverse careers and professions (noncomputer science or information technology fields such as medicine, movies, politics, art, etc.).
14. Ability to recognize uses of computing for creativity and self-expression.
15. Ability to recognize uses of automated data analysis to enhance understanding of complex natural and human systems.
16. Ability to describe how computing enhances communication, fostering new ways to communicate and collaborate.
17. Ability to evaluate how different forms of computing, such as cyber-physical systems and assistive technologies, enhance human capabilities.
18. Ability to apply computational thinking (as described above) to everyday life problems or actions.
19. Ability to recognize that computational (abstract) constructs can be applied to describe the functional properties of familiar objects/models.

Computing and Society

20. Ability to analyze the following effects of computing on societies within different economic, social, and cultural contexts:

    20.a. Ability to describe issues of equity and access for subgroups in the context of computing resources and computing fields.

    20.b. Ability to evaluate positive and negative impacts that the Internet and the web have had on societies and subgroups.

    20.c. Ability to evaluate legal and ethical concerns raised by computing-enabled innovation.

21. Ability to describe how the cultural context and audience for which computing artifacts are developed impacts their design and use.

Additional Focal Knowledge, Skills, and Attributes

- Knowledge of terminology and specific types of computers or computer parts
- Knowledge of specific computer innovations
- Knowledge of other domains/disciplines, including career knowledge
- Knowledge of social, cultural, ethical, and legal factors
- Knowledge of networks and information exchange

Characteristic Features

- All tasks must involve the use of computers or a computing task.
- All tasks must involve human use of and interaction with the computer or computing task.
- All tasks must involve networked systems.
- All tasks must involve a real-world scenario.
- All tasks must involve a societal or personal feature, such as economic, social, or cultural.
- All tasks must have an influence on one characteristic: privacy, legal, or ethical.

Variable Features

- Degree to which task requires knowledge of the internal working of a computer
- Whether the task requires description, comparison, or evaluation of computers or computing task
- Type of context provided (everyday life/professional life) and the familiarity of the discipline or context provided
- Type of issue that is being presented: legal, ethical, privacy, or other
- The audience or culture being presented in the problem
- Whether the problem is asking for benefits or drawbacks (positive presentation or negative presentation)

Potential Observations

- Appropriateness and/or completeness of the description of the use
- Appropriateness and/or completeness of the description/explanation of the use in contexts
- Appropriateness and/or completeness of the description of the innovation

- Appropriateness and/or completeness of the benefits and issues
- Appropriateness and/or completeness of the description of the effect of the design

Potential Work Products

- Recognize computing or computational solutions in everyday life
- Description of the use of computers and their component parts
- Description/explanation of how computers and computing are used (in various contexts)
- Description of a computing innovation (e.g., netwokring and the Internet)
- Description or explanation of the benefits or issues with the computing innovation, including examples
- Description of how context affects the design of a computing innovation

Analyze the Effects of Developments in Computing
Example Application: Computational Thinking in STEM

Data Analysis | **Modeling & Simulation** | **Computational Problem Solving** | **Systems Thinking**

Computational thinking is making inroads into science and mathematics classrooms, especially through the use of games, models, and simulations. The Computational Thinking in STEM (CT-STEM) project at Northwestern University is developing up to 60 computational thinking activities designed to be offered as part of existing STEM courses for high school students. These embedded activities address the computational thinking and modeling requirements in the NGSS and seek to impact a broad set of students through the vehicle of (at present) physics, chemistry, mathematics, and biology courses. Tools used include the agent-based simulation tool NetLogo, PhET simulations, Desmos math tools, and others.

The project has defined four areas of CT-STEM practices: Data Analysis (from collecting through analyzing and visualizing data); Modeling & Simulation (using, assessing, and designing models), Computational Problem Solving (see below), and Systems Thinking. Evidence-centered design has previously been applied to analyze model-based reasoning (Mislevy, Riconscente, & Rutstein, 2009) and systems thinking (Cheng, Ructtinger, Fujii, & Mislevy, 2010), and sssessment development for CT-STEM practices could draw on this work. For the CT-STEM computational problem solving practice, our design pattern on analyze the effects of developments in computing is applicable. CT-STEM's Computational Problem Solving practice consists of:

- Preparing problems for computational solutions
- Programming
- Choosing effective computational tools
- Assessing different approaches/solutions to a problem
- Developing modular computational solutions
- Creating computational abstractions
- Troubleshooting and debugging.

The CT-STEM work exemplifies FKSAs from the design pattern on analyzing the effects of developments in computing. Students who have completed the CT-STEM activities should show evidence of the following FKSAs from this design pattern:

7.a. Ability to describe an example of how computer programs are used to process information to gain insight and knowledge (including the use of computing tools to work with and make sense of large volumes of data in ways that humans cannot).

12. Ability to recognize when computing or computational solutions are applied to other (noncomputer science or information technology) disciplines.

18. Ability to apply computational thinking (as described above) to everyday life problems or actions.

FKSAs from other design patterns would apply to this work as well. For example, the CT-STEM Computational Problem Solving assessment has students decipher code to select outputs from among several choices. This relates to the design patterns analyze their computational work and the work of others and design and implement creative solutions and artifacts. CT-STEM activities could be mapped to these FKSAs, and then the ECD process could be used to develop assessments to measure CT-STEM practices.

Analyze Their Computational Work and the Work of Others

Overview

This design pattern supports the development of tasks in which students demonstrate that they can evaluate computational work (resulting in artifacts such as a program, program outputs, a website, or problem solution) and compare multiple computational artifacts. Students are able to recognize how different techniques can be used to solve problems or achieve computational goals in different ways. Students show they can evaluate the artifact in terms of its sufficiency (does it meet requirements?), accuracy (does it operate properly?), efficiency (does it use computing resources wisely?), and elegance (does it follow guidelines of modularity, parsimony, adherence to language-specific and language-general coding idioms, simplicity, well-structuredness, and comprehensibility?). They will demonstrate an understanding of how the forms that computers can take (including robotics) and their user interfaces affect usability (a special criterion for user-facing artifacts).

Focal Knowledge, Skills, and Attributes

Identify and Create Criteria for Evaluating Computational Work

1. Ability to describe the goal and/or outcome of a computational artifact (such as a program, website, or problem solution) and given requirements for the implementation (such as cost and delivery platform).
2. Ability to develop multiple evaluation criteria for a computational artifact (such as a program, website, or problem solution) along many dimensions (e.g., general goal/desired outcome, implementation method, intended users, context of use).
3. Ability to describe how form and design affect usability in human-computer interaction.

Evaluate Computational Work with Respect to Criteria (either holistically/black box or analytically)

4. Ability to evaluate computational work (such as a program, website, or problem solution) based on evaluation criteria including accuracy, sufficiency, efficiency, and elegance.
    - 4.1. Ability to identify errors in computational work (such as a program, website, or problem solution).
    - 4.2. Ability to evaluate the appropriateness of a computational artifact for an intended audience or user(s), including:
        - 4.2.1. All requirements are met, or competing requirements are properly prioritized,
        - 4.2.2. Accessibility (i.e., the inclusive practice of removing barriers that prevent interaction with, or access to, computational artifacts by people with disabilities) has been considered, and
        - 4.2.3. All possible inputs are considered.
    - 4.3. Ability to recognize that a working computational artifact could be implemented differently to optimize different criteria (e.g., more efficient in terms of memory use, speed, or more comprehensible computational approaches such as occur when refactoring code).
    - 4.4. Ability to evaluate a computational artifact based on aesthetics in the implementation design of the artifact (including simplicity, well-structuredness, modularity, and comprehensibility).

4.5. Ability to evaluate an artifact based on aesthetics in the look and feel of the computational artifact (user interface and usability).

Compare Multiple Solutions Along Criteria

5. Ability to evaluate trade-offs among multiple solutions (as computational artifacts) to the same problem based on some evaluation criteria (such as sufficiency, efficiency, accuracy, and elegance).

Additional Focal Knowledge, Skills, and Attributes

- Knowledge of hardware and software components of a computer system
- Knowledge of relevant evaluation criteria for computational solutions, such as speed, memory usage, power requirements, reusability of code, etc.
- Knowledge of representations of computational artifacts

Characteristic Features

- Tasks should provide the student with one or more examples of computational work (aka, artifact).
- The computational artifact should have obvious criteria on which it can vary, including different potential implementations and different usability.
- Tasks should communicate the goal or intended use of the artifact.

Variable Features

- Number of examples of computational work given
- Amount of detail given about the computational work
- Type of computational work and apparentness of the criteria
- Amount of detail provided about the evaluation criteria
- Difficulty level of discerning criteria (errors, usability, performance)
- Whether the artifact is yours or another's
- The degree to which the artifacts are usable vs. unusable

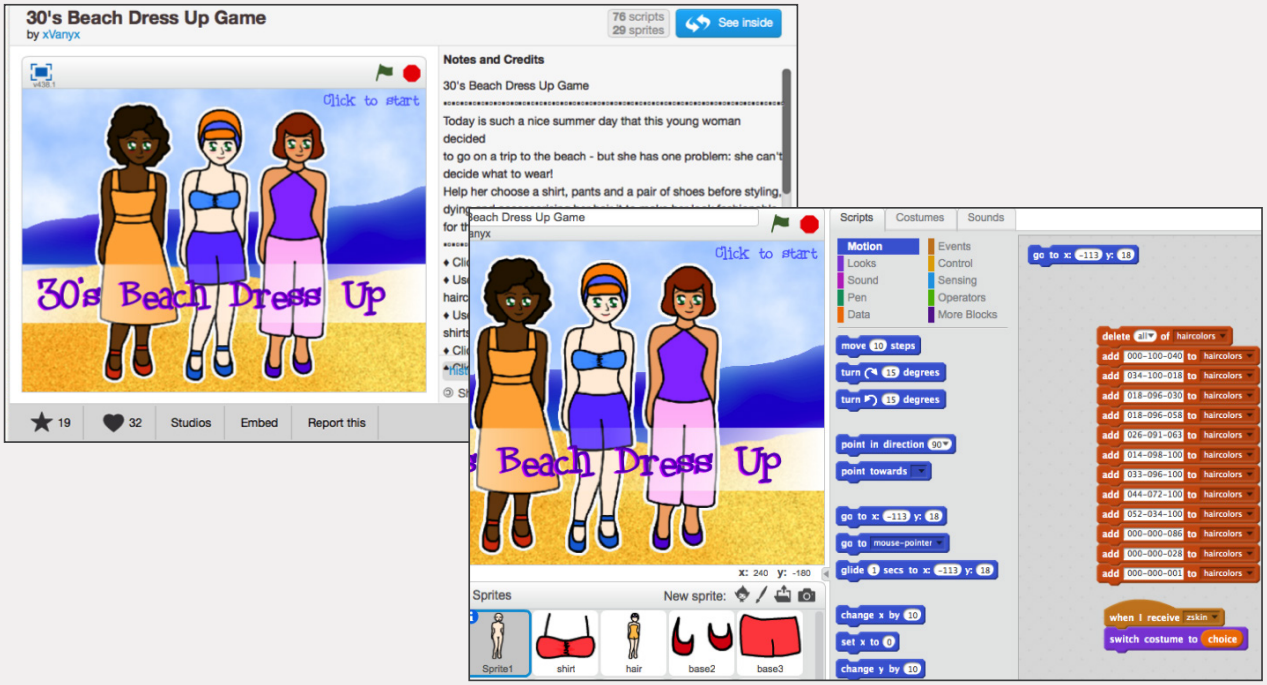
Potential Observations

- Appropriateness and thoroughness of the descriptions, evaluations, and comparisons of computational artifacts using different criteria
- Degree to which responses consider differences in requirement priorities, trade-offs among criteria, and different approaches to implementation
- Appropriateness of criteria to evaluate computational artifacts. Breadth of the dimensions considered
- Appropriateness of goal described for a computational artifact including references to requirements
- Degree to which the explanations relate to the goal of the artifact

Potential Work Products

- Descriptions, evaluations, and comparisons of computational artifacts in text, diagrams, etc., using both quantitative and qualitative criteria
- Descriptions of evaluation criteria and of dimensions
- Description of a goal for a computational artifact including references to requirements
- Explanations of why criteria were chosen for evaluation and comparison

Analyze Their Computational Work and the Work of Others
Example Application: Remixing in Scratch



Source: https://scratch.mit.edu/projects/69988504/

Computational thinking is not just for problem solving: It also finds a home in environments that support creativity and self-expression. In the context of "constructionist" pedagogies, the Scratch (https://scratch.mit.edu) environment provides a visual canvas for users to translate blocks of code into characters, actions, and sounds on the screen. The Scratch project website enables creators to share their stories, simulations, and art with others. When shared, the code to create these becomes available.

Based on the common programming practices of sharing and building on others' work, Scratch supports "reusing and remixing" as a computational thinking practice that puts coders into authentic contexts for understanding code and attributing ownership and enables creators to potentially build more complex code than they are currently capable of. Other Scratch computational practices include being incremental and iterative, testing and debugging, and abstracting and modularizing.

Reusing and remixing draw on the following FKSAs in the analyze computational work design pattern:

1. Ability to describe the goal and/or outcome of a computational artifact (such as a program, website, or problem solution) and given requirements for the implementation (such as cost and delivery platform).

4. Ability to evaluate computational work (such as a program, website, or problem solution) based on evaluation criteria including accuracy, sufficiency, efficiency, and elegance.

FKSAs from other design patterns would apply to this work as well. For example, there are legal and ethical issues around reuse, so FKSAs from analyze effects is applicable:

20.c. Ability to evaluate legal and ethical concerns raised by computing-enabled innovation.

Design and Apply Abstractions and Models

Overview

Thinking strategically about abstraction is a hallmark of computational thinking. This design pattern supports the development of tasks in which students use ideas and representations that capture general to specific aspects, or patterns, of an entity or a process and the relationships/structures among entities or processes, including level of detail. This may include designing general solutions to problems or generalizing a specific solution to encompass a broader class of problems (functional abstraction). These ideas and representations may be used in different contexts (problem or disciplines). Students demonstrate knowledge of the representational properties of discrete mathematics, models, diagrams, computer programs (data abstraction), items found in the natural and man-made world, and others. They also demonstrate an understanding of the limitations of models to represent phenomena and attention to the purpose of the model or abstraction.

Focal Knowledge, Skills, and Attributes

Foundational Knowledge Involving Abstraction in Computing

1. Ability to explain what abstraction is, both functional and data.
2. Ability to reason about a problem at multiple levels of detail.
3. Ability to explain the benefits of using abstraction in problem solving, e.g., to manage complexity and generalize patterns.
4. Ability to explain that an algorithm is a form of abstraction that contains a sequence of instructions whose end state or output can be determined once given a particular starting state.
5. Ability to explain the characteristics of problems for which abstraction would be useful.
6. Ability to describe how a computer model makes a representation of the real world.
7. Ability to explain how computers represent mathematical objects and logical operations for purposes of computation and modeling.
8. Ability to explain how computers represent objects as data and data as objects (e.g., media files, QR codes).
9. Ability to explain the connections between elements of mathematics and computer science including binary numbers, logic, sets, and functions.

Analyze a Model or Abstraction

10. Ability to analyze data to identify patterns through modeling and simulation.
11. Ability to evaluate how a model or simulation helps abstract a phenomenon and to computationally understand the phenomenon, e.g., how it reacts to various inputs.
12. Ability to explain why an abstraction is appropriate (or not appropriate) based on the purpose(s) of the abstraction.

    12.a. Ability to describe what features of an entity or process are captured in an abstraction and how an abstraction captures the essential features of a problem or process for a given purpose.

12.b. Ability to explain the value of abstraction (e.g., hiding details) to manage problem complexity for a given purpose.

12.c. Ability to analyze whether an abstraction is appropriate for a given purpose because functionally important factors/features were captured or omitted (e.g., in a model or simulation).

12.d. Ability to recognize or analyze whether an abstraction is appropriate for a given purpose because the right assumptions were made to accurately simplify a problem and make it easier to solve.

Create Abstraction or Models as a Mapping Between an Artifact and/or Model

13. Ability to generalize patterns across similar problems or processes.

14. Ability to break down or decompose a problem or process into parts or subproblems to simplify the complex.

15. Ability to explain the use and purposes of application program interfaces (APIs) and libraries to facilitate complex programming solutions by hiding details.

16. Ability to explain how parameterization can be used to generalize a specific solution.

17. Ability to describe how predefined functions, classes, and methods are used to divide a complex problem into simpler parts.

Design Solution

18. Ability to design an abstraction to represent a problem or solution.

19. Ability to create algorithms that capture features, aspects, characteristics, relationships, and/or structure of problems that meet specified purposes.

20. Ability to explain how the features of an algorithm map to features of a problem.

21. Ability to create solutions to problems at multiple levels of detail, including describing what components are at each level and relationships among components and between levels.

22. Ability to use abstraction as a means of separating specification from implementation.

23. Ability to express a solution, using standard design tools, that captures the relationships among entities or processes in the solution, and/or general to specific aspects of the entities or processes in the solution.

Analyze Using Computational Modeling, Simulation, and Data Manipulation

24. Ability to apply a variety of analysis techniques to find useful patterns in large data sets including text corpora and data that can be plotted in various ways.

25. Ability to use computers to test hypotheses about data.

26. Ability to describe ways in which computers use models of intelligent behavior (e.g., robot motion, speech and language understanding, and computer vision).

27. Ability to identify/describe how abstraction is used in reasoning about digital data (including how data are represented and how binary sequences can be interpreted)

Represent Alternatively

28. Ability to express abstractions using various representations, such as graphs, diagrams, cultural artifacts, storyboards, or computer programs.

29. Ability to use visual representations of problem states, structures, and data (e.g., graphs, charts, network diagrams, flowcharts).
30. Ability to represent data in a variety of ways including text, sounds, pictures, and numbers.
31. Ability to describe the features of large data sets that make them appropriate for automated analysis.
32. Ability to analyze communication (among people, computers, etc.) as forms of data exchange.

Additional Focal Knowledge, Skills, and Attributes

- Knowledge of mathematical concepts and representations
- Ability to code a solution to a problem
- Knowledge of various computational models
- Knowledge of specific algorithms
- Knowledge of the characteristics of an algorithm

Characteristic Features

- Must involve a computing task or problem where abstraction is appropriate.
- The task must be tied to a real-world situation.

Variable Features

- Is the abstraction given to the student, or something the student comes up with?
- The level of difficulty of the abstraction
- How far removed is the abstraction from the actual problem?
- The degree to which the abstraction involves mathematics (including logic)
- The representation (or type of representation) of the abstraction
- The type and complexity of the scenario given (includes not having a scenario)
- The number of situations provided in which students are asked to generalize
- The degree to which the problem given is decomposable
- Type of techniques appropriate to use for analysis of the problem space
- The size of the data set provided
- Degree to which the problem space is familiar
- The degree to which the problem space is well defined
- Group or individual work
- Correspondence of the elements between the abstraction and the problem
- Wether or not they talk about abstraction in general or as it applies to a particular problem

Potential Observations

- The degree to which the abstraction matches the need of the problem
- The accuracy of the representation
- The number of representations used
- The appropriateness of the representations

- The appropriateness of the explanation
- The degree to which the implementation matches the abstraction
- The degree to which abstraction is applied in the implementation
- The degree to which the map is appropriate for the problem and the elements
- The degree to which the explanation or documentation is a clear description of the abstraction (clarity of the explanation or documentation)
- The degree to which the analysis is appropriate
- The correctness of the analysis
- The appropriateness of the application or the correctness of the application of the abstraction
- The accuracy of the implementation or application of the abstraction

Potential Work Products

- One or more representations of an abstraction, problem, problem space, or analysis
- The explanation of or related to the abstraction (such as how it was applied, why it is appropriate)
- Implementation of the abstraction
- A mapping between elements of the problem and elements of the abstraction
- Explanation or documentation of the implementation or abstraction
- The analysis of the abstraction or model
- The application of the abstraction

Design and Apply Abstractions and Models
Example Application: Abstraction in Game Design

The Scalable Game Design (SGD) project uses a combination of tools, strong teacher education, and pedagogy to support student design of games and science simulations using the visual, drag-and-drop, agent-based language AgentSheets. Offered within existing computing education and STEM courses, the curriculum introduces students to computational thinking through game design and then advances to design of simulations in STEM topics (e.g., forest fire, predator-prey models). SGD advocates a "project-first" approach that pairs just-in-time skill *acquisition* with immediate skill *application* with the end goal of producing an executable and error-free tangible computational artifact.

SGD researchers have applied latent semantic analysis to thousands of student submissions to their game arcade and have discerned a common set of general patterns that describe agent/object interaction in games and simulations. These computational thinking patterns, such as collision, diffusion, path finding, and hierarchy of needs, are explicitly taught to students, and their use in programs is counted as evidence of increasing sophistication (Repenning, Webb, & Ioannidou, 2010).

SGD's patterns are good examples of the use of abstraction in understanding and solving problems through design in a particular context. They exemplify the following FKSAs:

2. Ability to reason about a problem at multiple levels of detail.
3. Ability to explain the benefits of using abstraction in problem solving, e.g., to manage complexity and generalize patterns.
12. Ability to explain why an abstraction is appropriate (or not appropriate) based on the purpose(s) of the abstraction.
20. Ability to explain how the features of an algorithm map to features of a problem.

Using these FKSAs and the SGD activities, we could instantiate the abstraction and modeling design pattern to produce assessment tasks that probe for student's deeper understanding of these patterns and how they serve as abstractions.

Design and Implement Creative Solutions and Artifacts

Overview

This design pattern supports the development of tasks in which students translate novel ideas and problem solutions into computational solutions and artifacts (i.e., artifacts that involve computing, such as robotics, and/or solutions that are implemented as computer programs). Students design and implement according to a given purpose or intent, including for creative expression. (In this way, programming can be viewed as a communicative act.) This design pattern encompasses steps of both problem solving and creative processes, including understanding, decomposing, exploring (e.g., by creating different representations of the problem with storyboards, flowcharts, and pseudocode), creating products that show one or more designed solutions and/or artifacts, and testing and improving the solution and or artifact.

Focal Knowledge, Skills, and Attributes

Frame/Explore the Problem/Approach

1. Understand that computational solutions can be designed for multiple purposes, e.g., a practical, personal, or societal aspect, personal curiosity, or expression of creativity.
2. Ability to describe creative aspects of the process used to design a computational artifact and of the artifact itself.
3. Ability to state a problem/approach in terms of what is given and what the end result or outcome should be.
4. Ability to evaluate existing programs and solutions (including students' own creations) in light of a new problem or purpose.
5. Ability to state what a program currently in design will output as a result of anticipated inputs.
    5.a. Ability to identify boundary conditions (edge cases) and how the program should handle them.
6. Ability to iterate on the statement of the problem/approach.

Decompose the Problem/Approach and Design the Solution/Artifact

7. Ability to classify problems as tractable, intractable, or computationally solvable/unsolvable.
8. Ability to break down a problem/intent into subparts in order to make it more manageable or comprehensible.
9. Ability to break down a problem with the goal of collaboratively building a solution with a team.
10. Ability to compose a solution by combining subparts that will lead to a solution to the problem as a whole.
11. Ability to identify what the logical and illogical inputs are for a computational solution.
12. Ability to identify boundary conditions that must be kept in mind when generating a computational solution.
13. Ability to design a computational solution that handles the desired range of inputs and is able to deal with boundary conditions/edge cases.

14. Ability to generate multiple approaches to solving a problem.
15. Ability to compare multiple approaches to solving a problem.
16. Ability to iteratively refine the design of the computational solution (based on results from implementation).
17. Ability to create a specification for an implementation of a solution or artifact.

Implement the Solution/Artifact

18. Understand how different tools (e.g., programming languages and software development environments), approaches (e.g., recursion), and methods (e.g., to describe a solution to a problem in words or representation separate from a program) could be used to create an effective solution to a problem.
19. Ability to evaluate what computational tool among a given set will be most appropriate to create a computational artifact.
20. Ability to compare the trade-offs between multiple approaches/techniques to implementing a problem based on certain evaluation criteria. For example, a program could be solved more elegantly (albeit more complex to code) through recursion, or a section of code could be made into a function or method (this may require advance planning, but the code becomes more modular), or use of a certain data structure may take more memory but reduce execution time, etc.
21. Ability to code complete solutions to problems (that handle the desired range of inputs and are able to deal with edge cases) in a programming language using an iterative process of coding, testing, and debugging.
22. Ability to use programming constructs to create executable solutions (i.e., code) including:
    - 22.a. Ability to use sequence in algorithmic instructions,
    - 22.b. Ability to use conditionals to incorporate different pathways based on selection criteria,
    - 22.c. Ability to use Boolean logic expression to incorporate selection and looping criteria, and
    - 22.d. Ability to incorporate repetition of blocks of code using looping constructs.
23. Ability to explain the alignment between a particular computational artifact and its specifications (including where the artifact differs from the specification of the problem or approach).

Test/Debug/Improve the Solution/Artifact

24. Ability to efficiently identify the source of run-time error(s).
25. Ability to explain the cause(s) of run-time error(s).
26. Ability to describe a systematic method for error-detection (such as test cases, unit testing, white box, black box, and integration testing).
27. Ability to implement testing and debugging methods to test and fix a computational solution.

Additional Focal Knowledge, Skills, and Attributes

- Knowledge of terminology
- Knowledge of specific programming languages

- Knowledge of specific design environments
- Knowledge of other domains/disciplines

Characteristic Features

- A problem or situation requiring a computational solution must be presented.

Variable Features

- Level of difficulty of the computational solution required
- Type of computational solution
- Type of problem provided
- Representation of the computational solution asked for
- Whether or not the computational solution is presented or asked for
- Degree to which the computational solution addresses the problem/situation/requirements of the solution
- Whether or not the student is coming up with a solution/evaluation a solution/or comparing multiple solutions

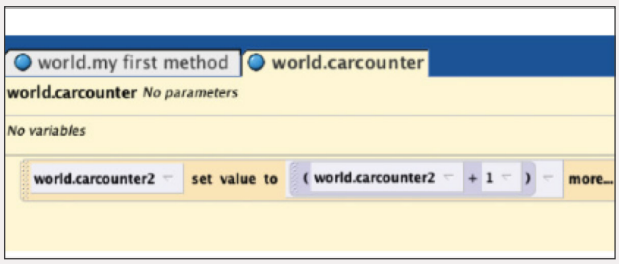
Potential Observations

- Degree to which the identified purpose is related to the computational solution
- Accuracy of the description of the design process
- Completeness of the description of the design process
- Degree to which the computational solution addresses the problem
- Flexibility in the computational solution
- Level of complexity of the computational solution
- Correctness of the computational solution (e.g., the degree to which it provides the expected output given input, the degree to which it runs without errors)
- Appropriateness of the use of programming structures in the computational solution
- Efficiency of the computational solution
- Accuracy of the explanation or description of the computational solution
- Completeness of the explanation or description of the computational solution (or degree to which the explanation or description covers the aspects of the computational solution being asked about)
- Accuracy of the description of the problem and problem space
- Completeness of the description of the problem and problem space (or degree to which the description covers the aspects of the problem and/or problem space being asked about)
- Accuracy of the explanation of the different tools
- Completeness of the description of the debugging process (or degree to which the description covers the aspects of the debugging process being asked about)
- Efficiency of the debugging process
- Degree to which the debugging process found and/or corrected the errors
- Degree to which the comparison addresses the differences between the two solutions or strategies
- Accuracy of the comparison of the solutions or strategies

Potential Work Products

- Identification of the purpose (or purposes) of the computational solution
- Description of the design process
- The computational solution
- Description or explanation of the computational solution
- Description of the problem and problem space (includes a description of the subparts to a problem and boundary conditions)
- Explanation of how different tools may be used or were used to create an artifact
- Description of the debugging process (either as applied to a particular computational solution or in general)
- Trace of the debugging process
- Comparison of multiple computational solutions or strategies

Design and Implement Creative Solutions and Artifacts
Example Application: Game Creation in Alice



Game programming is increasingly being introduced in middle and high school as a way to design and implement creative computational artifacts that also engage students in complex problem solving. Using versions of Alice and Storytelling Alice, a team of researchers at ETR Associates and UC Santa Cruz has spent more than a decade studying children and game programming in middle school, with a focus on girls and ethnic minority youth who are historically underrepresented in computing.

The games created by the middle school children typically include game mechanics such as guessing, collecting, and hiding objects. The games rely on a range of programming constructs from simple event handlers to (student-created) methods, conditional statements, loops, variables (including lists), and Boolean logic. Students use several programming patterns in the course of creating these games, such as creation of a "Counter" integer variable, initializing and incrementing it, and making a change in the game state depending on the value of the variable.

Game programming draws on the following FKSAs in the *Design and Implement Creative Solutions and Artifacts* design pattern:

1. Understand that computational solutions can be designed for multiple purposes, e.g., a practical, personal, or societal aspect, personal curiosity, or expression of creativity.
10. Ability to compose a solution by combining subparts that will lead to a solution to the problem as a whole.
11. Ability to identify what the logical and illogical inputs are for a computational solution.
16. Ability to iteratively refine the design of the computational solution (based on results from implementation).
17. Ability to code complete solutions to problems (that handle the desired range of inputs and are able to deal with edge cases) in a programming language using an iterative process of coding, testing, and debugging.
22. Ability to use programming constructs to create executable solutions (i.e. code) including:
    - 22.a. Ability to use sequence in algorithmic instructions,
    - 22.b. Ability to use conditionals to incorporate different pathways based on selection criteria,
    - 22.c. Ability to use Boolean logic expression to incorporate selection and looping criteria,
    - 22.d. Ability to incorporate repetition of blocks of code using looping constructs.

FKSAs from other design patterns such abstraction and modeling also apply to game programming.

Communication and Collaboration in Computational Thinking

These two practices not only cross the other computational thinking practices we are modeling (e.g., you can communicate and collaborate as part of designing and implementing creative solutions and artifacts), but also are inherently performance based. The design patterns that we have created for these crosscutting computational thinking practices provide guidance on what to look for as part of communication or collaboration. While communication and collaboration are important skills (e.g., OECD, 2013), teachers are not always given guidance in a curriculum on how to teach and assess them. For example, in our work with ECS, we found that the collaboration component is underspecified in terms of stated learning objectives. We found only one ECS objective that relates directly to collaboration, although there are collaborative projects listed as activities in the curriculum. (Like other objectives, these may be addressed more in the ECS teacher professional development than in the objectives stated in the curriculum as written.) Our design of the communication and collaboration design patterns were inspired by work on applying ECD to scenario-based learning. Related work is examining approaches for assessing content-oriented skills (such as our computational thinking practices) in a collaborative framework assessed in the context of work that would involve these practices (Davier & Halpin, 2013).

Communicate Thought Processes and Results

Overview

Communicating about computational artifacts supports many phases of computational thinking. This design pattern supports the development of tasks in which students show that they can communicate the process and results of their work in a way that is appropriate for the particular audience. Students can articulate major themes and ideas related to computing in writing and orally supported by graphs, visualizations, and computational analysis.

Computer Science Principles

- Explanation of the meaning of results
- Description of the impact of a technology or artifact
- Summarization of the behavior of a computational artifact
- Explanation of the design of an artifact
- Description of technology or artifact
- Justification of the appropriateness and correctness

Focal Knowledge, Skills and Attributes

1. Ability to generate written or visual documentation (e.g., report, essay, presentation slides, video) that describes a computational artifact, computational problem or intent, problem solution, or process to support the development of a computational artifact or problem solution (includes program documentation).

2. Ability to use different representations (such as storyboards, graphs, flowcharts, and code documentation) to communicate ideas about computational artifacts, computational problems/intents, or computational problem solutions/designs.
3. Ability to describe a computational artifact, computational problem or intent, problem solutions, or process of developing a computational artifact or problem solution for different purposes (e.g., including all key information for an audience for purposes such as making a decision or evaluating consequences.)
4. Ability to organize, explain, and present information about computational artifacts and solutions orally so it is clear to (that is, adapted to) a given technical or nontechnical audience. Special audiences include:
   - 4.a. Users
   - 4.b. Team members
   - 4.c. Adapters/other developers
5. Ability to use vocal, behavioral, and visual cues to clearly express ideas to an audience and engage the audience in the content.

Additional Knowledge, Skills, and Attributes

- Understanding of technical vocabulary in the domain (domain specific language)
- Knowledge of appropriate strategies for communicating an idea: vocal, behavioral, and visual cues

Variable Features

- Intended audience could be technical or nontechnical

Collaborate with Peers on Computing Activities

Overview

This design pattern supports the development of tasks in which a student demonstrates the ability to engage in the collaborative aspects of computer science by jointly solving problems and designing/creating computational artifacts. Students shows they can work with peers, experts, and others using collaborative practices such as pair programming and working in project teams, including groups or pairs.

A broader definition comes from the PISA Collaborative Problem Solving Framework:

> Collaborative problem-solving or collaborative design competency is the capacity of an individual to effectively engage in a process whereby two or more agents attempt to solve a problem or design an artifact by sharing the understanding and effort required to come to a solution/design and pooling their knowledge, skills and efforts to reach that solution/design." (Adapted from PISA, 2013, pg. 7).

Computer Science Principles

- Application of effective teamwork practices.
- Collaboration of participants.

- Production of artifacts that depend on active contribution from multiple participants.
- Documentation describing the use, functionality, and implementation of an artifact.

Focal Knowledge, Skills, and Attributes

1. Ability to discuss and develop a shared understanding of a problem and requirements in a given scenario.
2. Ability to proactively seek and collect solutions and discuss alternative solutions to a problem within a group or pair.
3. Ability to provide accurate, understandable, and tactful feedback to team members.
4. Ability to understand, value, and accept multiple perspectives on a problem and its solution(s).
5. Ability to integrate feedback from multiple perspectives to develop a solution to a computational thinking problem (e.g., code reviews, discussing specifications with clients, usability testing).
6. Ability to collaborate/participate across different time spans and locations.
7. Ability to use collaborative tools (e.g., Dropbox auto-sync, Google docs) and techniques (e.g., pair programming, code reviews).

INTERPERSONAL

8. Ability to work through and resolve conflicts among ideas and people.
9. Ability to work toward consensus and trust building with other members of the team.
10. Ability to ensure that everyone has a role within the team and that all necessary tasks are covered/that together the tasks accomplish the overall goal.
11. Ability to take on useful roles within a team, i.e., one that improves the understanding of and outcomes for the problem/intent/approach.
12. Ability to recognize and build on expertise distributed across the team.

Additional Knowledge, Skills and Attributes (Additional KSAs)

- Willingness to put in the effort to work with another student.
- Ability to identify how collaboration influences the design and development of software products.
- Ability to communicate (see other Design Pattern)
- Ability to identify what makes an effective team/pair and reflect on how well a team worked together, e.g., cooperation, not competition; contribution, not holding back.
- Ability to find the best partner (for computing work)
- Ability to set up physical space to support effective collaboration (e.g., room for both students at monitor)

Potential Observations

- Joint attention: students are looking at the same things
- Mutual engagement: students are actively involved with each other
- Individual agency: each student has responsibility and opportunity for action in the team and for learning from the team's work
- Group action and accountability: students are discussing, making, or problem solving together

- Group Organization: students have designed/taken on roles, responsibilities, and measures of progress
- Constructive discourse patterns: asking high-level questions to advance the understanding of the group, making and acknowledging contributions, finding common ground, providing and receiving help, etc.
- Monitoring and reflecting on teamwork (i.e., meta-cognition and self-regulation)

Characteristic Features

- Computational task is complex enough for a team to tackle and for multiple inputs to be incorporated.

Variable Features

- Team members: peers, experts, users.
- Location of team members

Use of Design Patterns

When domain modeling is complete, the assessment argument is further described as specifications for tasks and assessments, evaluation procedures, and measurement models (Exhibit 2, Conceptual Assessment Framework layer). Often, the emphasis is on task development to help place students in observable performance situations that elicit the desired FKSAs. Observation of task performance can take many forms: paper-and-pencil tests; analysis of artifacts produced (Brennan & Resnick, 2012; Werner, Denner, & Campe, 2014); artifact-based interviews (Barron et al., 2002; Grover, Pea, & Cooper, 2015); strategies, tactics, and moves in game play (Kerr & Chung, 2012; Shute, 2011); student moves in simulation-based and tutored learning environments (Gobert et al., 2013); environments that make students' thinking visible (Linn, Clark, & Slotta, 2003); and, in more recent work, learning analytics-enabled environments that reveal cognitive and noncognitive factors that help or hinder student learning (Berland, Martin, Benton, Smith, & Davis, 2013). Evidence is not obtained from passively observing but by deliberately putting students in situations, challenges, or tasks that will elicit the needed evidence (Grover & Bienkowski, forthcoming).

At the conceptual assessment framework layer, FKSAs are selected (and possibly combined) to formulate the student model. Assessment designers decide which FKSAs the tasks will cover and what modifications (if any) need to be done to make them align with particular learning objectives in a curriculum. In a companion model, the evidence model, designers define how tasks are going to be scored, what measurement models will be applied to the scores, and the meaning of the scores. Then the combination of the two models defines precisely what will be inferred about the student's performance based on the assessment.

Designers also define the task model at the conceptual assessment framework layer. The task model specifies the number and types of tasks to be included and describes the specific requirements of the assessment, such as the format of the items (e.g., paper and pencil) and the amount of time a student has to complete it. Note that the student, evidence, and task models are not created in a strict linear manner; rather, the assessment designers refine the models in conjunction, iterating among them to bring them into alignment and prepare a logically coherent foundation for the new assessment. Thus, as the task model is created, it may influence the earlier work on the student and evidence models.

Finally, at the assessment implementation and delivery layers, items and assessment forms are developed, reviewed, and validated in accordance with standards for validity (AERA/APA/NCME, 2014). For example, early piloting work can include think-alouds with students as they complete the tasks, expert reviews, and pilot testing with a sample of students close to the target population to calibrate the item difficulty and determine whether differences exist among subgroups of students (e.g, differential item functioning analyses, Osterlind & Everson, 2009). Later field testing may involve larger and more diverse samples of students and further psychometric work would be conducted at that point.

Summary and Next Steps

Our experience, as well as others' (Haertel, et al., in press-a), is that a rigorous and principled approach to assessment design yields not only valid assessments of well-defined knowledge and skills, but also templates for new sets of assessments that can be used to measure the same knowledge or skills in new contexts. The design patterns described in this report represent a first step to building such templates so that computational thinking practices, collaboration, and communication can be assessed in a wide variety of learning environments and with different delivery formats.

For use in assessment development, these design patterns should be crossed with the learning objectives of the curriculum to be assessed. When developing assessment tasks, there are additional areas to consider. For one, the design patterns described in this report do not currently specify the knowledge that is entailed in computational thinking. This is in part because the type of knowledge needed may depend on the context for which the skills are being taught. Therefore, when developing an assessment for a particular content area it is important to identify the knowledge that may go along with the practices. This can help in the development of items that determine whether students are struggling with the practice or with the knowledge. For building assessments of practices, the domain-specific knowledge may serve as an additional rather than a focal knowledge or skill. Such items would provide classroom teachers with diagnostic information.

Our work on these practices was anchored in the decisions made in the AP Computer Science Principles and ECS curricula to adopt four core practices and two crosscutting practices as key constructs in introductory computer science in secondary schools. These practices do not represent comprehensive knowledge and skills across the computer science domain but instead represent core practices, in the spirit of going against the mile-wide, inch-deep coverage in previous K-12 science standards. And like the newest standards for science and mathematics, they assess the application of design and inquiry skills to solve computational problems. When appropriate, the FKSAs in the practices can be assessed in the context of collaboration and communication; alternatively, they can be assessed independently.

Evidence-centered design gives us confidence about our argument from evidence when tasks are used to measure learning. Design patterns help us both explain what is important to measure and provide guidelines for how to approach measuring what is important.

As previously noted, the creation of design patterns is a necessarily interdisciplinary and iterative process. The design patterns presented in this report are a principled, vetted but initial step in modeling the computational thinking practices domain for assessment development purposes. The design patterns are thus works in progress, and we expect to further engage the computer science education and assessment communities in their evolution. The most up-to-date versions will always be available on our website, pact.sri.com.

References and Bibliography

2020 Computing [Special issue]. (2006). *Nature, 440*(7083). doi:10.1038/440413a

Adams, J. B. (2008). Computational science as a twenty-first century discipline in the liberal arts. *Journal of Computing Sciences in Colleges, 23*(5), 15–23.

Aho, A. V. (2011). Ubiquity Symposium: Computation and Computational Thinking. *Ubiquity, 2011(*January). doi:10.1145/1922681.1922682

Alvarado, C., Dodds, Z., & Libeskind-Hadas, R. (2012). Increasing women's participation in computing at Harvey Mudd College. *ACM Inroads, 3*(4), 55–64.

American Educational Research Association (AERA), American Psychological Association (APA) & National Council on Measurement in Education (NCME). (2014). *The Standards for Educational and Psychological Testing*. Washington, DC: AERA. Retrieved December 5, 2015 from http://www.apa.org/science/programs/testing/standards.aspx

Armoni, M. (2009). Reduction in computer science: A (mostly) quantitative analysis of reductive solutions to algorithmic problems. *JERIC, 8*(4), 1–30.

Arpaci-Dusseau, A., Astrachan, O., Barnett, D., Bauer, M., Carrell, M., Dovi, R., Franke, B., Gardner, C., Gray, J., Griffin, J., Kick, R., Kuemmel, A., Morelli, R., Muralidhar, D., Osborne, R. B., & Uche, C. (2013). *Computer science principles: Analysis of a proposed advanced placement course*. Paper presented at the 44th ACM Technical Symposium on Computer Science Education, Denver, CO. doi:10.1145/2445196.2445273

Astrachan, O., Hambrusch, S., Peckham, J., & Settle, A. (2009). T*he present and future of computational thinking*. Paper presented at the Proceedings of the 40th ACM Technical Symposium on Computer Science Education, Chatanooga, TN. doi: 10.1145/1508865.1509053

Astrachan, O., & Briggs, A. (2012). The CS Principles Project. *ACM Inroads, 3*(2), 38–42. doi:10.1145/2189835.2189849

Barr, V., & Stephenson, C. Bringing computational thinking to K-12: What is involved and what is the role of the computer science education community? *ACM Inroads, 2(*1), 48-54. doi:10.1145/1929887.1929905.

Barr, D., Harrison, J. & Conery. L. (2011). Computational Thinking: A Digital Age Skill for Everyone. *Learning & Leading with Technology, 38*(6) 20-23. Retrieved from http://csta.acm.org/Curriculum/sub/CurrFiles/LLCTArticle.pdf

Barron, B., Martin, C., Roberts, E., Osipovich, A., & Ross, M. (2002). Assisting and assessing the development of technological fluencies: Insights from a project-based approach to teaching computer science. *Proceedings of the Conference on Computer Support for Collaborative Learning,* Boulder, CO. 668–669.

Basawapatna, A., Koh, K. H., Repenning, A., Webb, D. C., & Marshall, K. S. (2011). *Recognizing computational thinking patterns*. Paper presented at the 42nd ACM Technical Symposium on Computer Science Education. doi:10.1145/1953163.1953241.

Basawapatna, A. R., Repenning, A., & Lewis, C. H. (2013). *The simulation creation toolkit: an initial exploration into making programming accessible while preserving computational thinking*. Paper presented at the 44th ACM Technical Symposium on Computer Science Education (501-506). doi:10.1145/2445196.2445346

Basu, S., Dickes, A., Kinnebrew, J.S., Sengupta, P., & Biswas, G. (2013). *CTSiM: A Computational Thinking Environment for Learning Science through Simulation and Modeling*. Paper presented at the 5th International Conference on Computer Supported Education, Aachen, Germany. Retrieved December 1, 2015 from http://www.teachableagents.org/papers/2013/Basu_ICCE.pdf

Bennedssen, J., & Caspersen, M. E. (2008, September). Abstraction ability as an indicator of success for learning computing science? In *Proceedings of the Fourth international Workshop on Computing Education Research* (pp. 15-26). ACM.

Bennett, V. E., Koh, K., & Repenning, A. (2013). *Computing Creativity: Divergence in Computational Thinking*. Paper presented at the 44th ACM Technical Symposium on Computer Science Education. Denver, CO. doi:10.1145/2445196.2445302

Berdik, C. (2015, November 24). *What a School District Designed for Computational Thinking Looks Like*. KQED Mind/Shift. Retrieved from http://ww2.kqed.org/mindshift/2015/11/24/what-a-school-district-designed-for-computational-thinking-looks-like/

Berland, M., Martin, T., Benton, T., Petrick Smith, C., & Davis, D. (2013). Using Learning Analytics to Understand the Learning Pathways of Novice Programmers. *Journal of the Learning Sciences, 22*(4), 1–36. doi:10.1080/10508406.2013.836655

Bienkowski, M. & Snow, E. (2014). *Building evidence and building practice in Computer Science education*: White paper presented at the 2014 Future Directions in Computer Science Education Summit. Orlando, FL. Retrieved from https://stacks.stanford.edu/file/druid:mn485tg1952/BienkowskiMarieSRI.pdf

Bienkowski, M., Rutstein, D., & Snow, E. (2015). *Computer Science Concepts in the Next Generation Science Standards*. Presented at the 2015 annual meeting of the American Educational Research Association (AERA), Chicago, IL.

Bienkowski, M. (2015). Making Computer Science a First-Class Object in the K-12 Next Generation Science Standards (Abstract Only). In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15)*. ACM, New York, NY, p. 513.

Bootstrap. (2015). Bootstrap materials: curriculum and software. Retrieved December 5, 2015 from http://www.bootstrapworld.org/materials/fall2015/index.shtml

Bouvier, D., Chen, T.-Y., Lewandowski, G., McCartney, R., Sanders, K., & VanDeGrift, T. (2012). User Interface Evaluation by Novices. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education* (pp. 327–332). New York, NY, USA: ACM. doi:10.1145/2325296.2325372

Brennan, K., & Resnick, M. (2012). *New frameworks for studying and assessing the development of computational thinking*. Paper presented at the 2012 Annual Meeting of the American Educational Research Association, Vancouver, Canada.

Buffum, P. S., Lobene, E. V., Frankosky, M. H., Boyer, K. E., Wiebe, E. N., & Lester, J. C. (2015). A Practical Guide to Developing and Validating Computer Science Knowledge Assessments with Application to Middle School. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (pp. 622–627). New York, NY, USA: ACM. doi:10.1145/2676723.2677295

Camp, T. (2012). "Computing, we have a problem…." *ACM Inroads, 3*(4), 34–40. doi:10.1145/2381083.2381097

Cápay, M., & Magdin, M. (2013). Alternative methods of teaching algorithms. *Procedia-Social and Behavioral Sciences, 83,* 431–436.

Chazelle, B. (2012). Natural Algorithms and Influence Systems. *Communications of the ACM, 55*(12), 101–110. doi:10.1145/2380656.2380679

Cheng, B. H., Ructtinger, L., & Fujii, R. (2010). *Assessing Systems Thinking and Complexity in Science* (*Technical Report No. 7).* Menlo Park, CA: SRI International. Retrieved from https://www.sri.com/work/publications/assessing-systems-thinking-and-complexity-science

Colburn, T., & Shute, G. (2007). Abstraction in computer science. *Minds and Machines, 17*(2), 169–184.

Cortina, T. (2007). An Introduction to Computer Science for Non-Majors Using Principles of Computation. *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education* (pp. 218-222). doi:10.1145/1227310.1227387

Danielsiek, H., Paul, W., & Vahrenhold, J. (2012). Detecting and understanding students' misconceptions related to algorithms and data structures. *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (pp. 21-26). doi: 10.1145/2157136.2157148

Davier, A. A., & Halpin, P. F. (2013). *Collaborative problem solving and the assessment of cognitive skills: Psychometric considerations.* ETS Research Report Series. Retrieved November 15, 2015 from https://www.ets.org/Media/Research/pdf/RR-13-41.pdf

DeBarger, A. H., & Snow, E. (2010). *Design Pattern On Model Use In Interdependence Among Living Systems (Large-Scale Assessment Technical Report 13)*. Menlo Park, CA: SRI International. Retrieved from https://www.sri.com/work/publications/design-pattern-model-use-interdependence-among-living-systems-large-scale-assessme

della Cava, M. (2015). Should students learn coding? Students, schools disagree, poll finds. USA Today. Retrieved from http://www.usatoday.com/story/tech/2015/08/20/google-gallup-poll-finds-parents-want-computer-science-education-but-administrators-arent-sure/31991889/

Denning, P. J. (2009). The Profession of IT beyond computational thinking. *Communications of the ACM, 52*(8), 28-30.

Dwyer, H., Hill, C., Carpenter, S., Harlow, D., & Franklin, D. (2014). Identifying Elementary Students' Pre-Instructional Ability to Develop Algorithms and Step-By-Step Instructions. *Proceedings of the 45th ACM Technical Symposium on Computer Science Education* (pp. 511-516). doi: 10.1145/2538862.2538905

Fan, L. (2012, July). Learning of algorithms: A theoretical model with focus on cognitive development. *The 12th International Congress on Mathematics Education,* Seoul, Korea. Retrieved from http://eprints.soton.ac.uk/358300/

Forišek, M., & Steinová, M. (2012). Metaphors and Analogies for Teaching Algorithms. *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (pp. 15–20). doi:10.1145/2157136.2157147

Friend, M., & Cutler, R. (2013, August). *Efficient egg drop contests: How middle school girls think about algorithmic efficiency.* Paper presented at the International Computing Education Research Conference, San Diego, CA.

Futschek, G. (2006). Algorithmic thinking: The key for understanding computer science. In R. T. Mittermeir (Ed.), *Informatics education–The bridge between using and understanding computers.* (pp. 159–168). Berlin, Germany: Springer-Verlag

Gal-Ezer, J., & Zur, E. (2002). The concept of "algorithm efficiency" in the high school CS curriculum. *Proceedings of the 32nd ASEE/IEEE Frontiers in Education Conference*. Boston, MA.

Gal-Ezer, J., & Zur, E. (2003). The efficiency of algorithms—Misconceptions. *Computers & Education, 42*(3), 215–226. doi:10.1016/j.compedu.2003.07.004.

Gal-Ezer, J., & Stephenson, C. (2010). Computer science teacher preparation is critical. *ACM Inroads, 1(*1), 61–66.

Gallup. (2015). *Searching for Computer Science: Access and Barriers in U.S. K-12 Education*. Gallup. Retrieved from http://csedu.gallup.com/183725/landscape-computer-science-education.aspx

Garcia, D. D., Harvey, B., & Segars, L. (2012). CS principles pilot at University of California, Berkeley. *ACM Inroads, 3*(2), 58-60. doi:10.1145/2189835.2189853

Garner, S., Haden, P., & Robins, A. (2005). My program is correct but it doesn't run: A preliminary investigation of novice programmers' problems. In A. Young & D. Tolhurst (Eds.), *Proceedings of the 7th Australasian Conference on Computing Education*, Vol. 42 Australian Computer Society, Inc.. Darlinghurst, Australia. (pp. 173–180).

Gibson, P. J. (2012, July). T*eaching graph algorithms to children of all ages*. Paper presented at the Annual Conference on Innovation and Technology in Computer Science Education, Haifa, Israel.

Ginat, D., & Menashe, E. (2015). SOLO Taxonomy for Assessing Novices' Algorithmic Design. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (pp. 452–457). doi:10.1145/2676723.2677311

Gobert, J., Sao Pedro, M., Raziuddin, J., and Baker, R. S., (2013). From log files to assessment metrics for science inquiry using educational data mining. *Journal of the Learning Sciences. 22*(4), 521-563.

Goldman, K., Gross, P., Heeren, C., Herman, G., Kaczmarczyk, L., Loui, M. C., & Zilles, C. (2008). Identifying important and difficult concepts in introductory computing courses using a delphi process. *ACM SIGCSE Bulletin, 40*(1), 256-260.

Goode, J., Chapman, G., & Margolis, J. (2012). Beyond curriculum: The Exploring Computer Science program. *ACM Inroads, 3*(2). doi:10.1145/2189835.2189851

Goode, J., & Margolis, J. (2011). Exploring computer science: A case study of school reform. *ACM Transactions on Computing Education, 11*(2), 1–16. doi:10.1145/1993069.1993076

Google for Education. (n.d.). *Exploring Computational Thinking: CT Overview*. Retrieved December 5, 2015, from www.google.com/edu/resources/programs/exploring-computational-thinking/

Grover, S., & Pea, R. (2013). Computational Thinking in K–12: A review of the state of the field. *Educational Researcher, 42*(1), 38-43. doi:10.3102/0013189X12463051

Grover, S., & Pea, R. (2012). Using a discourse-intensive pedagogy and Android's App Inventor for introducing computational concepts to middle school students. *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (pp. 723-728). doi: 10.1145/2445196.2445404

Grover, S. (2013, May 28). Learning to Code Isn't Enough. Retrieved from https://www.edsurge.com/news/2013-05-28-opinion-learning-to-code-isn-t-enough

Grover, S., Pea, R., & Cooper, S. (2015). *"Systems of Assessments" for Deeper Learning of Computational Thinking in K-12.* Paper presented at the Annual Meeting of the American Educational Research Association, Chicago, IL.

Grover, S., Pea, R., & Cooper, S. (2015). Designing for deeper learning in a blended computer science course for middle school students. *Computer Science Education, 25*(2), 199–237. doi:10.1080/08993408.2015.1033142

Grover, S., & Bienkowski, M. (forthcoming). *Process over product for studying computational thinking practices in introductory programming.* Accepted for presentation at the Annual Meeting of the American Educational Research Association, Washington DC.

Guzdial, M. (2008). Paving the way for computational thinking. Communications of the *ACM 51*(8), 25-27.

Haertel, G. D, Vendlinki, T. P., Rutstein, D., DeBarger, A., Cheng, B. H., Snow, Eric B., D'Angelo, C., Harris, C., Yarnall, L., & Ructtinger, L. (in press-a). General introduction to evidence-centered design. In H. Braun (Ed.), *Meeting the challenges to measurement in an era of accountability* (pp. 107-148). London, UK: Routledge

Haertel, G. D., Vendlinski, T. P., Rutstein, D., DeBarger, A., Cheng, B. H., Ziker, C., Harris, C. J., D'Angelo, C., Snow, E. B., Bienkowski, M., & Ructtinger, L. (in press-b). Assessing the life sciences: Using evidence-centered design for accountability purposes. In H. Braun (Ed.), *Meeting the challenges to measurement in an era of accountability.* London, UK: Routledge.

Hasni, T. F., & Lodhi, F. (2011). Teaching Problem Solving Effectively. *ACM Inroads, 2*(3), 58–62. doi:10.1145/2003616.2003636

Hazzan, O., Lapidot, T., & Ragonis, N. (2011). *Guide to teaching computer science: An activity-based approach*. New York, NY: Springer.

Hazzan, O. (2003). How Students Attempt to Reduce Abstraction in the Learning of Mathematics and in the Learning of Computer Science. *Computer Science Education, 13*(2), 95–122. doi:10.1076/csed.13.2.95.14202

Hazzan, O. (2008). Reflections on Teaching Abstraction and Other Soft Ideas. *ACM SIGCSE Bulletin, 40*(2), 40–43. doi:10.1145/1383602.1383631

Hazzan, O., & Kramer, J. (2007). Abstraction in Computer Science & Software Engineering: A Pedagogical Perspective. *Frontier Journal, 4*(1), 6–14. Retrieved from https://www.hwswworld.com/pdfs/frontier35.pdf

Hemmendinger, D. (2010). A plea for modesty. *ACM Inroads, 1*(2), 4-7.

Hershkowitz, R., Schwarz, B. B., & Dreyfus, T. (2001). Abstraction in Context: Epistemic Actions. *Journal for Research in Mathematics Education, 32*(2), 195–222. doi:10.2307/749673

Hill, J. H. (2007). Applying abstraction to master complexity: The comparison of abstraction ability in computer science majors with students in other disciplines. *Proceedings of the 2nd International Workshop on the Role of Abstraction in Software Engineering* (pp. 15-21). doi: 10.1145/1370164.1370169

Hu, C. (2011). Computational Thinking: What It Might Mean and What We Might Do About It. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education* (pp. 223–227). doi:10.1145/1999747.1999811

Jona, K., Wilensky, U., Trouille, L., Horn, M., Orton, K., Weintrop, D., & Beheshti, E. (2014). *Embedding Computational Thinking in Science, Technology, Engineering, and Math (CT-STEM)*. Presented at the Future Directions in Computer Science Education Summit, Orlando, FL. Retrieved from http://ccl.northwestern.edu/papers/2014/OrtonKaiNorthwestern-1.pdf

Kaczmarczyk, L. C., Petrick, E. R., East, J. P., & Herman, G. (2010). Identifying student misconceptions of programming. *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (pp. 107–111). doi: 10.1145/1734263.1734299

Kafai, Y. B., Peppler, K. A., & Chapman, R. N. (Eds.). (2009). *The Computer Clubhouse: Constructionism and creativity in youth communities.* New York, NY: Teachers College Press.

Kerr, D., & Chung, K. W. K. (2012). Identifying key features of student performance in educational video games and simulations through cluster analysis. *Journal of Educational Data Mining, 4*(1), 144–182.

Koh, K.H., Basawapatna, A.R., Bennett, V.E., & Repenning, A. (2010) Towards the Automatic Recognition of Computational Thinking for Adaptive Visual Language Learning. *Proc. of VL/HCC '10*, IEEE Computer, Madrid, Spain, 59-66

Kramer, J. (2007). Is Abstraction the Key to Computing? *Communications of the ACM, 50*(4), 36–42. doi:10.1145/1232743.1232745

Lee, I., Martin, F., Denner, J., Coulter, B., Allan, W., Erickson, J., et al. (2011). Computational thinking for youth in practice. *ACM Inroads, 2*(1), 32-37. doi:10.1145/1929887.1929902

Lewis, C. M., & Shah, N. (2012). Building upon and enriching grade four mathematics standards with programming curriculum. *Proceedings of the 43rd ACM technical symposium on Computer Science Education* (pp. 57-62). doi: 10.1145/2157136.2157156

Linn, M. C., Clark, D., & Slotta, J. D. (2003). WISE design for knowledge integration. *Science Education, 87(*4), 517-538.

Lister, R. (2012). The CC2013 Strawman and Bloom's Taxonomy. *ACM Inroads, 3*(2), 12–13. doi:10.1145/2189835.2189840

Liu, M., & Haertel, G. (2011). *Design Patterns: A Tool to Support Assessment Task Authoring (Large-Scale Assessment Technical Report 11)*. Menlo Park, CA: SRI International. Retrieved from http://ecd.sri.com/downloads/ECD_TR11_DP_Supporting_Task_Authoring.pdf

Loman, N., & Watson. (2013). So you want to be a computational biologist? *Nature Biotechnology, 31*, 996-998. doi:10.1038/nbt.2740

Lopez, M., Whalley, J., Robbins, P., & Lister, R. (2008). Relationships Between Reading, Tracing and Writing Skills in Introductory Programming. *Proceedings of the Fourth International Workshop on Computing Education Research* (pp. 101–112). doi:10.1145/1404520.1404531

Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010). The Scratch Programming Language and Environment. *Trans. Comput. Educ., 10*(4), 16:1–16:15. doi:10.1145/1868358.1868363

Malyn-Smith, J., & Lee, I. (2012). Application of the Occupational Analysis of Computational Thinking-Enabled STEM Professionals as a Program Assessment Tool. *Journal of Computational Science Education, 3*(1), 2–10. Retrieved from http://www.shodor.org/media/content/jocse/volume3/issue1/jocse_volume3_issue1#page=8

Margolis, J., Ryoo, J. J., Sandoval, C. D. M., Lee, C., Goode, J., & Chapman, G. (2012). Beyond Access: Broadening Participation in High School Computer Science. *ACM Inroads, 3*(4), 72–78. doi:10.1145/2381083.2381102

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y., B-D., Laxer, C., Thomas, L., Utting, I., & Wilusz, T. (2001). *A multi-national, multi-institutional study of assessment of programming skills of first-year CS students.* ACM SIGCSE Bulletin. 33(4). 125-180. doi: 10.1145/572139.572181

Meerbaum-Salant, O., Armoni, M., & Ben-Ari, M. (2011). Habits of Programming in Scratch. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education* (pp. 168–172). doi:10.1145/1999747.1999796

Meerbaum-Salant, O., Armoni, M., & Ben-Ari, M. (Moti). (2013). Learning computer science concepts with Scratch. *Computer Science Education, 23*(3), 239–264. doi:10.1080/08993408.2013.832022

Messick, S. (1994). The interplay of evidence and consequences in the validation of performance assessments. *Educational Researcher, 23*(2), 13-23.

Microsoft Corporation. (2014). *Computational Thinking in the Sciences and Beyond.* Retrieved from http://research.microsoft.com/apps/video/dl.aspx?id=232450

Mislevy, R., & Riconscente, M. (2005). *Evidence-centered assessment design: Layers, structures, and terminology (PADI Technical Report 9)*. Menlo Park, CA: SRI International. Retrieved from https://www.sri.com/work/publications/evidence-centered-assessment-design-layers-structures-and-terminology-padi-technic

Mislevy, R. J., & Riconscente, M. M. (2006). Evidence-centered assessment design: Layers, concepts, and terminology. In S. Downing & T. Haladyna (Eds.), *Handbook of Test Development* (pp. 61-90). Mahwah, NJ: Erlbaum

Mislevy, R. J., Steinberg, L. S., & Almond, R. G. (2003). *On the Structure of Educational Assessments.* Los Angeles, CA: Center for the Study of Evaluation, National Center for Research on Evaluation, Standards, and Student Testing, Graduate School of Education & Information Studies, University of California, Los Angeles.

Mislevy, R. J., Riconscente, M. M., & Rutstein, D. W. (2009). *Design Patterns for Assessing Model-Based Reasoning (Large-Scale Assessment Technical Report 6)*. Menlo Park, CA: SRI International. Retrieved from http://ecd.sri.com/downloads/ECD_TR6_Model-Based_Reasoning.pdf

Muller, O., Ginat, D., & Haberman, B. (2007). Pattern-oriented Instruction and Its Influence on Problem Decomposition and Solution Construction. In Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (pp. 151–155). New York, NY, USA: ACM. doi:10.1145/1268784.1268830

Muller, O., & Haberman, B. (2009). *A Course Dedicated to Developing Algorithmic Problem Solving Skills - Design and Experiment*. Presented at the PPIG 2009 - 21st Annual Workshop, Limerick, Ireland. Retrieved from http://www.ppig.org/workshops/ppig-2009-21st-annual-workshop

Murphy, L., Fitzgerald, S., Lister, R., & McCauley, R. (2012). Ability to "Explain in Plain English" Linked to Proficiency in Computer-based Programming. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research* (pp. 111–118). New York, NY, USA: ACM. doi:10.1145/2361276.2361299

National Governors Association (NGA) Center for Best Practices, Council of Chief State School Officers. (2010). *Common Core State Standards*. Washington, DC: National Governors Association Center for Best Practices, Council of Chief State School Officers.

National Research Council. (2012). *A Framework for K-12 Science Education: Practices, Crosscutting Concepts, and Core Ideas*. (Committee on a Conceptual Framework for New K-12 Science Education Standards. Board on Science Education, Division of Behavioral and Social Sciences and Education, Ed.).Washington, DC: The National Academies Press. Retrieved from http://www.nap.edu/catalog/13165/a-framework-for-k-12-science-education-practices-crosscutting-concepts

National Research Council (2010). *Report of a workshop on the scope and nature of computational thinking*. Washington, DC: The National Academies Press. Retrieved from http://www.nap.edu/.

National Research Council. (2012). *Report of a workshop on the pedagogical aspects of computational thinking.* Washington, DC: The National Academies Press. Retrieved from http://www.nap.edu/.

NGSS Lead States. (2013). *Next Generation Science Standards: For States, By States*. Washington, DC: National Academy of Sciences.

Nicholson, K., Good, J., & Howland, K. (2009). C*oncrete thoughts on abstraction*. Paper presented at the Psychology of Programming Interest Group (PPIG 2009) Limerick, Ireland . Retrieved from http://www.ppig.org/library/paper/concrete-thoughts-abstraction

Northwestern University. (2015). Computational Thinking in STEM: Lesson Plans. Retrieved December 5, 2015, from http://ct-stem.northwestern.edu/lesson-plans/

OECD. (2013, March). PISA 2015: *Draft Collaborative Problem Solving Framework*. Retrieved from http://www.oecd.org/pisa/pisaproducts/Draft%20PISA%202015%20Collaborative%20Problem%20Solving%20Framework%20.pdf

Osterlind, S. J., & Everson, H. T. (2009). *Differential item functioning (2nd ed., Vol. 161)*. Thousand Oaks, CA: SAGE Publications.

Petersen, A., Craig, M., & Zingaro, D. (2011). Reviewing CS1 Exam Question Content. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education* (pp. 631–636). doi:10.1145/1953163.1953340

Phillips, P. (2009). Computational Thinking: a problem-solving tool for every classroom. Retrieved from http://www.csta.acm.org/Resources/sub/ResourceFiles/CompThinking.pdf

Qin, H. (2009). Teaching computational thinking through bioinformatics to biology students. *Proceedings of the 40th ACM Technical Symposium on Computer Science Education* (pp. 188-191). doi: 10.1145/1508865.1508932

Razzouk, R., & Shute, V. (2012). What Is Design Thinking and Why Is It Important? *Review of Educational Research (82)*3, 330-348. doi:10.3102/0034654312457429

Regev, A., & Shapiro, E. (2002). Cellular abstractions: Cells as computation. *Nature, 419*, 343-419. doi:10.1038/419343a

Repenning, A., Webb, D., & Ioannidou, A. (2010). Scalable Game Design and the Development of a Checklist for Getting Computational Thinking into Public Schools. *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (pp. 265–269). doi:10.1145/1734263.1734357

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Kafai, Y. (2009). Scratch: Programming for All. *Communications of the ACM, 52*(11), 60–67. doi:10.1145/1592761.1592779

Robertson, J., & Howells, C. (2008). Computer game design: Opportunities for successful learning. *Computers & Education, 50*(2), 559-578.

Rutstein, D., Snow, E., & Bienkowski, M. (2014). *Computational thinking practices: Analyzing and modeling a critical domain in computer science education*. Paper presented at the 2014 annual meeting of the American Educational Research Association (AERA), Philadelphia, PA.

Sakhnini, V., & Hazzan, O. (2008). Reducing abstraction in high school computer science education: The case of definition, implementation, and use of abstract data types. *Journal on Educational Resources in Computing (JERIC), 8*(2), 1–13.

Samurçay, R. (1989). The concept of variable in programming: Its meaning and use in problem-solving by novice programmers. In E. Soloway & J. C. Spohrer (Eds.), *Studying the novice programmer* (pp. 161–178). Mahwah, NJ: Lawrence Erlbaum.

Scalable Game Design: Computational Thinking Patterns. (2014, June 3). Retrieved from http://sgd.cs.colorado.edu/wiki/Category:Computational_Thinking_Patterns

Seehorn, D., Carey, S., Fuschetto, B., Lee, I., Moix, D., O'Grady-Cunniff, D., ... & Verno, A. (2011). CSTA K--12 Computer Science Standards: Revised 2011.

Sengupta, P., Kinnebrew, J.S., Basu, S., Biswas, G., & Clark, D. (2013). Integrating Computational Thinking with K-12 Science Education Using Agent-based Computation: A Theoretical Framework. *Education and Information Technologies, 18*(2), 351-380. doi:10.1007/s10639-012-9240-x

Shein, E. (2014). Should everybody learn to code? *Communications of the ACM, 57*(2), 16-18. doi:10.1145/2557447

Shute, V. J. (2011). Stealth assessment in computer-based games to support learning. *Computer Games and Instruction, 55*(2), 503–524.

Snow, E., Fulkerson, D., Nichols, P., & Feng, M. (2011). *Design patterns to support storyboards and scenario-based, innovative item types*. Paper presented at the 2011 annual meeting of the American Educational Research Association (AERA), New Orleans, LA.

Sudol-DeLyser, L. A. (2015). Expression of Abstraction: Self Explanation in Code Production. *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (pp. 272–277). doi:10.1145/2676723.2677222

Sykora, C. (2014, September 11). Computational thinking for all. Retrieved from https://www.iste.org/explore/articleDetail?articleid=152&category=Solutions&article=Computational-thinking-for-all

Szalay, A., & Gray, J. (2006). 2020 Computing: Science in an exponential world. *Nature, 440*, 413-414. . doi:10.1038/440413a

Tadmor, B., & Tidor, B. (2005). Interdisciplinary research and education at the biology-engineering-computer science interface: a perspective. *Drug Discovery Today, 10*(23-24), 1706-1712. doi:10.1016/S1359-6446(05)03702-5

Tew, E. A., & Guzdial, M. (2010). The FCS1: A Language Independent Assessment of CS1 Knowledge. *Proceedings of the The 42nd ACM Technical Symposium on Computer Science Education*. doi:10.1145/1953163.1953200

Tew, A. E., & Guzdial, M. (2010). Developing a Validated Assessment of Fundamental CS1 Concepts. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (pp. 97–101). doi:10.1145/1734263.1734297

Thompson, E., Luxton-Reilly, A., Whalley, J. L., Hu, M., & Robbins, P. (2008). Bloom's Taxonomy for CS Assessment. In *Proceedings of the Tenth Conference on Australasian Computing Education - Volume 78* (pp. 155–161). Darlinghurst, Australia, Australia: Australian Computer Society, Inc. Retrieved from http://dl.acm.org/citation.cfm?id=1379249.1379265

Tucker, A., Deek, F., Jones, J., McCowan, D., Stephenson, C., & Verno, A. (2003). A model curriculum for K–12 computer science: Final report of the ACM K-12 task force curriculum committee. New York, NY: The Association for Computing Machinery.

Vardi, M. Y. (2012). What is an Algorithm? *Communications of the ACM, 55*(3), 5–5. doi:10.1145/2093548.2093549

Walker, H. M. (2012). Developing a Useful Curricular Map. *ACM Inroads, 3*(4), 14–16. doi:10.1145/2381083.2381089

Werner, L., Campe, S., & Denner, J. (2012). Children Learning Computer Science Concepts via Alice Game-programming. *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (pp. 427–432). doi:10.1145/2157136.2157263

Werner, L., Denner, J., Bliesner, M., & Rex, P. (2009). *Can middle-schoolers use Storytelling Alice to make games? Results of a pilot study.* Paper presented at the 4th International Conference on Foundations of Digital Games. doi:10.1145/1536513.1536552

Werner, L., Denner, J., Campe, S., & Kawamoto, D. C. (2012). The Fairy Performance Assessment: Measuring Computational Thinking in Middle School. *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (pp. 215–220). doi:10.1145/2157136.2157200

Werner, L., Denner, J., & Campe, J. (2014). Children programming games: A strategy for measuring computational learning. *Transactions on Computing Education, 14*(4), 24:21–24:22. doi:10.1145/2677091

Wilensky, U., & Reisman, K. (2006). Thinking like a wolf, a sheep, or a firefly: Learning biology through constructing and testing computational theories—an embodied modeling approach. *Cognition and Instruction, 24*(2), 171-209. doi:10.1207/s1532690xci2402_1

Wilson, C., Sudol, L. A., Stephenson, C., & Stehlik, M. (2010). Running on empty: The failure to teach K-12 computer science in the digital age. A*ssociation for Computing Machinery/Computer Science Teachers Association.*

Wing, J. M. (2006). Computational Thinking. *Communications of the ACM, 49*(3), 33–35. doi:10.1145/1118178.1118215

Wing, J. M. (2008). Computational thinking and thinking about computing. *Philosophical Transactions of the Royal Society a: Mathematical, Physical and Engineering Sciences, 366*(1881), 3717–3725. doi:10.1098/rsta.2008.0118

SRI Education™

SRI Education, a division of SRI International, is tackling the most complex issues in education to identify trends, understand outcomes, and guide policy and practice. We work with federal and state agencies, school districts, foundations, nonprofit organizations, and businesses to provide research-based solutions to challenges posed by rapid social, technological and economic change. SRI International is a nonprofit research institute whose innovations have created new industries, extraordinary marketplace value, and lasting benefits to society.

Silicon Valley
(SRI International headquarters)
333 Ravenswood Avenue
Menlo Park, CA 94025
+1.650.859.2000
education@sri.com

Washington, D.C.
1100 Wilson Boulevard, Suite 2800
Arlington, VA 22209
+1.703.524.2053

www.sri.com/education

Stay Connected