Using Programming Case Studies to Foster Computational Thinking

Arthur D. Hanna, Ph. D.
AHanna@StMaryTX.edu

St. Mary's University, San Antonio, Texas 78228

ABSTRACT

Computational thinking is thinking like a computer scientist, the kind of thinking absolutely required to formulate computational solutions to algorithmic problems. Aristotle wrote, "For things we have to learn before we can do them, we learn by doing them." Which begs the question: How can a student do something that the student has not yet learned to do? Answer: Let the expert (teacher) do it with them, but provide a way for the student to actively participate in the process. Expert show-and-tell programs do not actively involve the student. The sink-or-swim approach, which requires the student to write large programs from scratch with little or no expert help, does actively involve the student, but "swimmers" are usually students who already can think computationally, whereas "sinkers" are students who cannot.

This paper's answer for fostering computational thinking, the programming case study, requires the student to fully understand the problem and to actively engage in re-solving it by answering computational thinking questions and by supplying "missing code" segments. A case study includes: 1) an introduction of background material required by the student to understand the statement of the problem and its solution; 2) a clear statement of the problem; 3) a sample of the program/user dialog to provide the student with test data; 4) a collection of computational/critical thinking questions which are keyed to the program listing; and 5) the program listing with code segments strategically elided which become the student's responsibility (aka, "missing code").

1. INTRODUCTION

1.1 Why is Computational Thinking So Important?

Wing (33) wrote, "Computational thinking is a fundamental skill for everyone, not just for computer scientists. To reading, writing, and arithmetic, we should add computational thinking to every child's analytical ability." So, it is not surprising to find articles like these in the popular press, "Why your 8-year-old should be coding" (O'Dell) and "Why You Need to Know Code" (Snow).

Wing (33) goes on to define computational thinking as "a way of solving problems, designing systems, and understanding human behavior that draws on concepts fundamental to computer science." The following excerpt from the 2010 NRC report on a workshop on computational thinking offers a little more detail

> …computational thinking includes a broad range of mental tools and concepts from computer science that help people solve problems, design systems, understand human behavior, and engage computers to assist in automating a wide range of intellectual processes. The elements of computational thinking are reasonably well known, given that they include the computational concepts, principles, methods, languages, models, and tools that are often found in the study of computer science…Concepts from computer science such as algorithm, process, state machine, task specification, formal correctness of solutions, machine learning, recursion, pipelining, and optimization also find broad applicability.

The website for Carnegie Mellon's Center for Computational Thinking states, "Computational thinking is a way of solving problems, designing systems, and understanding human behavior that draws on concepts fundamental to computer science. Computational thinking is thinking in terms of abstractions, invariably multiple layers of abstraction at once. Computational thinking is about the automation of these abstractions. The automation could be an algorithm, a Turing machine, a tangible device, a software system—or the human brain."

Computational thinking is the quintessential tool of the Computer Scientist. Unfortunately, computational thinking is not natural and it must be explicitly taught.

1.2 Programming Requires Algorithmic Thinking (and More)

The author believes programming should be required in every Computer Science course. Why? In 1989, the "Denning Report" (Denning) makes it very clear how fundamentally important algorithmic thinking is to Computer Science, "The discipline of computing is the systematic study of algorithmic processes that describe and transform information: their theory, analysis, design, efficiency, implementation, and application. The fundamental question underlying all of computing is: What can be (efficiently) automated?" Programming is not the only means available to the Computer Scientist for expressing computational thought, but it is arguably the primary one. What McCracken (2) wrote twenty years ago about how important computer programming is to the study of Computer Science is still just as true, "Computer Science is more than programming, but a Computer Science graduate must be able to program, at least."

Thankfully, at least for the more aesthetically inclined, programming is not all science (Gries); it is art as well. Niklaus Wirth describes programming as "a constructive art." Frederick Brooks, famous for his description of the joys and woes of programming a computer, aptly describes the value of the art of programming when he answers the question, "Why is programming fun?" The analysis, design, and testing phases of

programming are artistic because each requires a great degree of creativity if done correctly. This sense of creative artistic effort is the same sense used when describing problem solving as a creative process in (Papert) and (Polya).

1.3 Algorithmic Thinking is a Subset of Computational Thinking

Guzdial (26) writes that Alan Perlis "argued that programming was an exploration of process, a topic that concerned everyone, and that the automated execution of process by machine was going to change everything. He saw programming as a step toward understanding a 'theory of computation,' which would lead to students recasting their understanding of a wide variety of topics…in terms of computation."

Logical thinking, sequential thinking, procedural thinking, mathematical thinking are all aspects of computational thinking. Computational thinking is, at least, Perlis' "understanding…in terms of computation." What David Harel (12) called "algorithmics" Wing (33) generalized to "computational thinking" and called for computational thinking to be recognized as a fundamental skill for everyone, as important an analytic ability as reading, writing, and arithmetic.

1.4 Difference between Novice and Expert

Armstrong (2) writes, "Psychologists use the term *expert* to refer to an individual who is significantly more experienced than others in performing a particular task. However, the difference between experts and novices cannot be reduced solely to experience (time invested in learning how to perform tasks)." Herr (11) describes some of the most important differences between experts and novices which should motivate pedagogy for teaching computational thinking

1. Because of what experts already know, they recognize meaningful features and patterns that are not noticed by novices.
2. Experts rely on accumulated experience which manifests as wisdom or intuition. Experts are more flexible than novices because they have more strategies and more effective strategies for performing a task. Novices' dearth of experience forces them to rely on formal procedures for handling a task.
3. Experts think about how they organize, represent, and interpret information. Experts think about their thinking, they think critically, they meta-cogitate.
4. Experts have significant "chunks" of content knowledge organized around the core, big concepts of their discipline.
5. An expert's knowledge is not sets of isolated facts, but is "conditionalized" which means that the expert selectively retrieves knowledge only under specific circumstances. Experts build meaningful relations among related "chunks." The expert's overt organizational scheme provides a deep understanding of the discipline. Experts can see "the big picture" and are aware of the specific circumstances of the task at hand. As a result, experts can integrate new information more efficiently.

6. Experts can retrieve the important “chunks” of their knowledge with little explicit effort. Experts practice automaticity, that is, automatic or fluent retrieval. Novices are distracted by the cognitive processing they are obliged to remain conscious of during their performance of a task.

1.5 Learning by Using Case Studies

Novice computational thinkers need to learn how to think like experts to become good computational thinkers. Linn (123) suggests that the most effective way to learn to think like an expert is to emulate expert behavior. A novice could be apprenticed to an expert, but the logistics of apprenticeship for learning computational thinker is unworkable. This paper’s major premise is that an effective way to learn to emulate expert behavior is to “watch” it happen by engaging in case study.

Hanna (42) defines programming *case study* as a detailed exposition of how a specific problem is solved by an expert in the problem’s domain. A case study emphasizes pragmatics—the problem-specific common sense employed by an expert—and usually assumes the reader of the case study has already been formally introduced to declarative knowledge necessary for solving the case study’s problem. This declarative knowledge is both the facts about the problem itself and the problem’s domain of application and also the programming language’s syntax and semantics. As an expert analyzes a problem and synthesizes the problem’s solution, she asks precise questions of herself which she then attempts to answer in the most accurate, logical, and appropriate way she can. The *case study method* employs case studies to demonstrate to the novice the computational thinking experts use to solve problems. *Computational and critical thinking questions* attempt to make explicit the thought process used by the domain expert. The heuristics in Polya are a classic attempt to develop a generic collection of questions (both computational and critical thinking questions) to aid problem solving in all domains.

Most of the author’s students enjoy the case study approach and, more importantly, they benefit from it. The few students who complain do so because they find that carefully attending to the questions is hard work.

As part of his dissertation, the author began building a collection of case studies similar to the ones described by Clancy (1992a, 1992b) (a case study approach to teaching the Pascal programming language); Shapiro (the collection of case studies developed to help beginning programmers learn to program well); and the erstwhile “Literate Programming” columns in the *Communications of the ACM*.

1.6 Reading Source Code

A significant aspect of using case studies as described herein is that students are required to read-to-understand a significant amount of the author’s source code. Reading code for understanding has a number of benefits

1. Reading source code teaches the student to learn to read source code, a skill necessary for work in industry. All too often the existing documentation for a program in maintenance is faulty. As a result, source code is often the only real source of "the truth."
2. Reading masterful source code teaches the student good programming habits. The author agrees with Skorkin (2), "Reading great code is just as important for a programmer as reading great books is for a writer."
3. Reading source code that is already part of the solution of a big problem, in lieu of requiring the students to write all of the code themselves, allows the students to solve more and bigger problems.
4. Requiring students to read a large amount of source code usually has the side-effect of encouraging students to work in small code-reading groups.

2. PROGRAMMING CASE STUDY DESIGN

A programming case study includes 1) an introduction of background material required by the student to understand the statement of the problem and its solution; 2) a clear statement of the problem; 3) a sample of the program/user dialog to provide the student with test data; 4) a collection of computational thinking questions which are keyed to the program listing; and 5) the program listing with code segments strategically elided which become the student's responsibility (aka, "missing code").

Advice Create a single Word document which contains your complete case study

1. Introduction
2. Problem Statement
3. Sample Program Dialog
4. Computational/Critical Thinking Questions
5. Code (all of the source code, but with strategic code segments deleted, of course)

Use a portrait-oriented single section with your favorite variable-width, easy-to-read font (like Times Roman) for parts 1, 2, and 4, but use a fixed-width font (like Courier New) for the Sample Program Dialog. For the sake of readability, it is convenient to use a separate landscape-oriented section with line numbers and a fixed-width font for the Code.

Because all of the code is included in the case study's Word document, it is very easy to refer to the code in the Computational/Critical Thinking questions. Simple copy-and-paste operations from the Word document into the IDE the students use suffices to create the program's source files.

The author makes the case study document available in electronic form by posting it in the content portion of the course management software his university uses. Sometimes it is helpful to briefly survey the case study for the students on the day it is posted, but presenting details without giving students time to prepare usually proves to be fruitless. Require students to prepare the case study—read it and jot down questions about what

they do not understand—so that the case study can be formally presented during the next class meeting. Your formal presentation should be students-ask-questions/teacher-provides-answer and should engender some discussion of the particulars. Sometimes a quiz addressing a few strategically chosen computational and critical thinking questions serves to provoke student questions.

2.1 Picking a Problem to Solve

In her concluding paragraphs, Wing (33) advised, "Intellectually challenging and engaging scientific problems remain to be understood and solved. The problem domain and solution domain are limited only by our own curiosity and creativity."

One of the author's favorite lines is, "Inside the chest of every serious Computer Scientist beats the heart of a Mathematician!" There is a significant overlap between computational thinking and mathematical thinking. Wing (35) agrees when she writes, "Computer science inherently draws on mathematical thinking, given that, like all sciences, its formal foundations rest on mathematics." The author's Mathematics professor friends decry the shrinking mathematical thinking skills of university undergraduates in recent years. A problem chosen for a case study can easily be made to include a few mathematical concepts, so the computational and critical thinking questions can be opportunities to learn those concepts.

Advice Your problem can come from almost anywhere: the course text book, your research areas and those of his peers, SIGCSE Nifty Assignments, puzzles and games from newspapers, magazines, and books, simulation of "interesting" things or processes, programming contests, et cetera.

What is important, of course, is to find a problem that the students can "buy into." If a problem is too difficult, then the students will be too dependent on the professor's contribution to solving the problem. Try to find a problem that is on the frontier of the students' abilities thereby ensuring that the students are required to stretch to understand the professor's contribution to the solution and to develop their part of solution.

2.2 Solving the Problem

The author credits the development of his expertise over the past thirty years to his penchant for problem solving. New problems are very exciting to work on. Being required to learn a new language feature or to learn an entirely new programming language or to work in an entirely new problem domain or solve a problem to help a peer do research is a joy.

Advice *Watch* what you do as you solve the problem so that you notice both what and how you do your problem-solving. It is very easy to forget how difficult certain aspects of the solution were to develop and it is difficult to remember the subtleties of the problem-solving you have accomplished. Waiting until after you have finished problem-solving allows the questions you have asked yourself and the insights you have gained—

the questions you want your students ask and the insights you want your students to gain—to be lost.

The student audience is the most important consideration in how you solve the problem. Ask yourself, "What do I want to teach? To which audience?" Novice programmers tend to be more focused on particulars of the programming language and the IDE being used, while more experienced programmers should be more interested in software architecture, data modeling, and algorithmic design (elements of computational thinking). So, with the student audience in mind, you should take note of

1. the questions you ask of yourself
2. the mistakes that you make (and their consequences), the starts and stops, and the mistakes that you avoid (Niels Bohr defined an expert as "a person who has made all the mistakes that can be made in a very narrow field!")
3. the major decisions that you make during the problem-solving process
4. the tradeoffs you make
5. the simplifications and/or generalizations you are forced to make
6. the research you are required to do to fully understand the problem's functional requirements and the solution's non-functional requirements (programming language, run-time support, time complexity, et cetera)
7. how you can "play" with aspects of the programming language syntax and semantics to allow you to demonstrate some of the variety of ways to express parts of the solution

Above all, enjoy yourself. Another of the author's favorite lines is, "If you're not having a good time, then you must not be doing it right!" You are asking your students to participate vicariously with you as collectively you problem-solve your way toward a solution to the problem. If you, the master problem solver, cannot be passionate about the problem when you present it to the students, what makes you think that they will be passionate about it?

2.3 Writing the Introduction

A programming case study introduction is not always necessary. Some problems are so simply stated and easily solved that an introduction is superfluous, but it is very important to remember the knowledge and skills of the student audience.

Advice Ask yourself, "Can I remember how and when I came to understand what this case study is focused on? What do my students need to know to be able to understand the problem's domain, the description of the problem, and the code that I wrote to solve the problem?" For example, if the problem requires knowledge of combinatorial objects (permutations, combinations, and subsets), then ensure the student audience is comfortable enough with them to begin the case study. If the case study is coordinated with topics in the text book you are using for class, clearly there is no compelling reason for you to repeat what the student can easily find in their text book.

2.4 Writing the Description of the Problem

Inventor and engineer Charles F. Kettering once quipped, “A problem well stated is a problem half solved.” The statement of the problem should be long on description but very short on design.

Advice If your description seems excessively long, your introduction may be too sparse and/or you may be including too much solution in your problem statement.

As a general rule, case study problems should not be “canned” problems. It sounds a little unfair and perhaps a little counterintuitive, but the author believes it is good to leave the student with some questions about the problem. Real-world problems are usually not fully described at the onset. Telling all obscures the very natural repeating of analysis-to-requirements-to-design process that occurs when solving most big-enough, interesting problems. Students should be led to an understanding of the problem (see Section 2.6 below), not spoon fed all of its particulars. Requiring the students to incrementally come to understand the problem allows the students to realistically engage in the analysis and solution of the problem.

Ask yourself, “What, exactly, did I understand about the problem when I first encountered it? What were the key insights about the problem—not its solution, but its definition—that I had as I progressed toward the final solution?”

2.5 Providing Samples of Program/User Dialog

There are at least two good reasons for providing sample program/user interaction. The sample should contain input data and corresponding output data.

1. The student gets a better sense of what the program is expected to do when she is able to see how input data gets transformed into program output.
2. Since the student must re-create the solution by supplying missing code, it is important for them to know when their solution is correct.

This section of the case study is more accurately titled “Expected Program Output.” when the problem requires an answer which is unrelated to any user input.

Advice The sample you provide is not intended to be an exhaustive collection of test cases like those that might be found in a formal Test Plan. Students should be expected to develop their own test case and to perform their own testing to assure correct program execution.

In addition to generating a small sample of “normal” input/output combinations, consider generating anomalies (integer overflow, attempted division by zero, violation of preconditions, et cetera) to confront the student with the interesting and important “fringe” or boundary cases that they tend to ignore. Also, consider generating

instrumented output to provide students with insight into the dynamics of the program's execution.

When the sample is primarily text input and output, include the text in the case study document using a fixed-width font like Courier New. Include a graphic sample in the case study document as a screenshot image.

Sometimes a problem must process data stored in a separate file. If the file is not too large and if the contents are text, then consider providing the entire contents of the file as part of the case study document.

2.6 Writing the Computational/Critical Thinking Questions

From my dictionary, "question [Latin *quaerere* to ask, to seek] an interrogative sentence addressed to someone in order to get a reply." A statement records a fact or opinion—it supplies information—while a question seeks to obtain some missing piece of information.

Leeds (12) writes that there is obvious power in asking questions: questions demand answers, stimulate thinking, provide valuable information, and get people to persuade themselves (people are more apt to believe what they say, not what you say).

Experts employ their expertise when problem-solving by figuring out correct or reasonable answers to a certain body of problem-specific, domain-conditioned questions. To learn how to problem-solve, students must learn to think accurately and reasonably about the concepts that define the content and, because all content concepts are logically interdependent, to think through the connections between the concepts. More simply, students must to learn to ask questions like the experts do. The common-sense way to teach students what kind of questions to ask is to show them the questions the expert asks herself.

Advice Read through the case study from top to bottom—Introduction, Problem Statement, Sample Program Dialog, and Code—and ask yourself, "What am I trying to teach with this case study?" Also, keep in mind the notes you made when you originally solved the problem (see 2.2 Solving the Problem). Number the questions and, to make it easier for you and the students to establish the context for each question, ask the questions from top to bottom of the case study document.

Ask related questions as a separately numbered question, but preface the question with something like "7. (Continuing 6) ..."

When necessary to your discussion, "mark" portions of the case study in some fashion to emphasize them. The author uses shading, especially in the Code section, à la the prolific Deitel/Deitel father/son text book authors. It often aids the clarity of the question when you copy the highlighted portion of the case study into the question.

To help build student self-efficacy (Ramalingam), consider providing answers for a few questions (especially the harder questions); however, make the student "pay" for the answer by asking a question about the answer you provide.

The questions you ask should plumb the student's understanding of

1. aspects of the Introduction that need special emphasis (this is an excellent opportunity to teach about the problem's domain; for example, concepts from mathematics that are required and methods needed for modeling the problem's data)
2. aspects of the Problem Statement (especially when the description of the problem is purposely lacking in some important details)
3. inconsistencies and/or regularities in the Sample Program Dialog put there to teach about the program's dynamics
4. aspects of the solution process which do not manifest themselves
5. interesting aspects of the Code, including but not limited to
    a. unfamiliar syntax and semantics of the programming language
    b. the data modeling used to represent problem-domain objects and/or solution-domain objects
    c. the software architecture
    d. library functions used and/or eschewed
    e. justification for algorithms used and alternative algorithms considered
    f. time complexity of algorithms used
    g. hints for designing and coding of the missing code segments
    h. justification for design and code choices made by the case study author during her development
    i. software engineering considerations (programming style, anti-debugging, testing methods, parameterization, et cetera)

Very often you will find yourself trying to decide whether it makes sense to ask questions about a topic you covered in a case study earlier in the semester. You should consider ensuring that each case study is as independent of the semester context in which it originally created as possible. It is reasonable to repeat yourself. And remember, it never hurts for the students to revisit concepts.

You can learn much from the asking and answering of computational and critical thinking questions. It is pleasantly surprisingly how often, when revisiting or revising a programming case study, you discover how much you have learned since originally finding and solving the problem described therein. Correspondingly, it is unpleasantly surprising to find what a woeful job you did on your first attempt at constructing the case study. No matter how good of a job you think you've done, there is usually room for improvement. But remember, while a case study may become one of your "old friends," it is always a brand new acquaintance for most of your students when they meet it in your course.

2.7 Choosing Code Segments to Be Provided by Students

The author tells his students that his approach to selecting the code segments they then assume responsibility to re-create is to find the spots in the code where he was having the most "fun" during his problem-solving and then deleting them. The most "fun" code is almost always those segments which are the most hard-fought for, code that expresses new algorithms, or uses non-trivial data structures, or requires new-to-the-student language features.

Advice You should decide which code segments you want the students to re-create before you write most of the questions about the code (2.6 Writing the Computational/Critical Thinking Questions). Again, ask yourself, "What am I trying to teach with this case study?"

The author likens a student's case study work to a person who wears two caps! When wearing the "tool builder cap" the student's code typically completes development of modules that fit at the bottom of the structure chart where the utility, library-like functionality usually fits. But, when wearing the "tool user cap" the student's responsibility is typically to complete modules that occur toward the top of the structure chart where the problem-specific functionality is usually found. It's important to decide which "cap" you want your students to wear when working on any given case study.

For the large problems that case studies usually solve, it may be unfair to require beginning students to wear both "caps" at the same time. Of course, your students eventually will be required to wear both "caps" when they work in industry, so more mature students should be expected to do so when working on case studies in their upper-division courses.

As a general rule, do not require students to provide either the input or the output modules of an IPO-architected program, unless, of course, either or both are interesting enough to teach something new. If the main module is interesting, you may choose to elide parts of it. Certainly, some of or all of the processing modules (depending on your teaching intentions) are candidates for re-creation. If you do not allow students to modify the program's software architecture, then the students will be required to reuse module interfaces.

If the processing has a primary module that depends are several subordinate modules—especially if the primary module is complex—you may consider assigning only the subordinate modules to the students. Or you may assign a major portion of the primary module to the students and provide all of the subordinate modules.

Place a comment in place of the missing code segment which reads something like this, "Student provides missing code to…" The ellipsis must provide enough detail so that the student can combine that detail with knowledge of the problem and the answers to related computational/critical thinking questions to develop an algorithm for the missing code segment.

3. A PROGRAMMING CASE STUDY EXAMPLE (elided—ask author for details)

4. CONCLUSION

Computational thinking is thinking like a Computer Scientist and is essential to algorithmic problem solving. Unfortunately, computational thinking is not easy to teach. Students can teach themselves to think better when shown how by an expert. The programming case study method described in this paper is an effective way to "show" students how experts think.

The author's case studies are not copyrighted and are available for reuse and adaption.

5. REFERENCES

Armstrong. Notes on the Psychology of Expertise. Web. October 1, 2013. <*http://www.unm.edu/~jka/courses/archive/expertise.html*>.

Brooks, Frederick. The Mythical Man-Month: Essays on Software Engineering (3rd edition). *Addison-Wesley*. 1975. Print.

Clancy, Michael and Linn, Maria. Designing Pascal Solutions: A Case Study Approach. *New York: W.H. Freeman and Company*. 1992a. Print.

Clancy, Michael and Linn, Maria. Case Studies in the Classroom. *SIGCSE Bulletin*, 23:220-224. 1992b. Print.

Denning, Peter, et al. Computing as a Discipline. *Communications of the ACM*, 32:9-23. 1989. Print.

Gries, David. The Science of Programming. *New York: Springer-Verlag*. 1981. Print.

Guzdial, Mark. Paving the Way for Computational Thinking. *Communications of the ACM*, 51:25-27. 2007. Print.

Hanna, Arthur. *Using Case Studies to Teach the C Programming Language to Novice Programmers*. PhD thesis, UT Austin. 1996. Print.

Harel, David. Algorithmics: The Spirit of Computing. *Addison-Wesley*. 2004. Print.

Herr, Norman. Internet resources for The Sourcebook for Teaching Science. *John Wiley*. 2007. Print.

Leeds, Dorothy. The 7 Powers of Questions. *Berkley Publishing Group*. 2000. Print.

Linn, Maria and Clancy, Michael. The Case for Case Studies of Programming Problems. *Communications of the ACM*, 35:121-132. 1992. Print.

McCracken, David. Programming Languages in the Computer Science Curriculum. *SIGCSE Bulletin*, 24:1-4. 1992. Print.

National Research Council. Report of a Workshop on "The Scope and Nature of Computational Thinking." 2010. Print.

O'Dell, J. Why your 8-year-old should be coding. *VentureBeat*. Web. April 12, 2013.

Papert, Seymour. Mindstorms: Children, Computers and Powerful Ideas. *Basic Books, Inc*. 1980. Print.

Polya, George. 1957. How to Solve It. *Princeton: Princeton University Press*. Print.

Ramalingam, LaBelle, and Wiedenbeck. Self-efficacy and mental models in learning to program. *SIGCSE Bulletin*, 36:171-175. 2004. Print.

Shapiro, Henry. How to Program Well: A Collection of Case Studies. *Richard D. Irwin, Inc*. 1994. Print.

Skorkin, Alan. Why I Love Reading Other People's Code and You Should Too. Web. May 19, 2010. <http://www.skorks.com/2010/05/why-i-love-reading-other-peoples-code-and-you-should-too/>.

Snow, Shane. Why You Need To Know Code. Web. July 17, 2013. <http://www.linkedin.com/today/post/article/20130716014745-7374576-why-you-need-to-know-code-and-how-you-can-learn-in-a-month>.

Wing, Jeannette. Computational Thinking. *Communications of the ACM*, 49:33-35. Print.

Wirth, Niklaus. Algorithms + Data Structures = Programs. *Prentice-Hall, Inc*. 1976. Print.