Center for Brains, Minds & Machines

CBMM Memo No. 030

April 3, 2015

Towards a Programmer's Apprentice (Again)

by

Howard Shrobe Boris Katz Randall Davis
MIT CSAIL
32 Vassar Street
Cambridge, MA 02139

Abstract

Programmers are loathe to interrupt their workflow to document their design rationale, leading to frequent errors when software is modified—often much later and by different programmers. A Programmer's Assistant could interact with the programmer to capture and preserve design rationale, in a natural way that would make rationale capture "cost less than it's worth", and could also detect common flaws in program design. Such a programmer's assistant was not practical when it was first proposed decades ago, but advances over the years make now the time to revisit the concept, as our prototype shows.

This work was supported by the Center for Brains, Minds and Machines (CBMM), funded by NSF STC award CCF - 1231216.

Towards a Programmer's Apprentice (Again)

Howard Shrobe and **Boris Katz** and **Randall Davis**
MIT CSAIL
32 Vassar Street
Cambridge, MA 02139

Abstract

Programmers are loathe to interrupt their workflow to document their design rationale, leading to frequent errors when software is modified—often much later and by different programmers. A Programmer's Assistant could interact with the programmer to capture and preserve design rationale, in a natural way that would make rationale capture "cost less than it's worth", and could also detect common flaws in program design. Such a programmer's assistant was not practical when it was first proposed decades ago, but advances over the years make now the time to revisit the concept, as our prototype shows.

In the early 1970's, Chuck Rich, Dick Waters and Howard Shrobe, inspired by a number of other people around the MIT AI Laboratory (e.g., Terry Winograd, Carl Hewitt, Gerry Sussman), proposed the idea of a Programmer's Apprentice, an intelligent assistant that would help a programmer write, debug and evolve software (Rich et al. 1978; Rich and Shrobe 1976; 1978; Rich, Shrobe, and Waters 1979a; Rich and Waters 1981; Rich, Shrobe, and Waters 1979b; Rich and Waters 1988; 1990). Part of the vision was the idea that software systems always evolve over their lifetimes and that our failure to capture and preserve design rationale makes it difficult to extend and evolve systems without introducing new errors. But in practice, capturing rationale is often a burdensome and thankless task. Writing down the reason for a design choice involves a disruption of the normal flow of activity, one that causes the programmer to switch from programming to explaining and then back again. Furthermore, this effort is of benefit to somebody else, in the future. Even if that somebody else turns out to be yourself, the benefit is still in the (perhaps distant) future and its *current value* is minimal. To make rationale capture work, one has to **make it cost less than it's worth**.

To solve this problem, the apprentice was envisioned to be a knowledgeable junior programmer, one that would understand the basic patterns and clichés of programming and that would only ask questions about unusual aspects of a system's design, ones that the programmer would actually be willing to chat about. These unusual elements might involve not only mistakes and oversights but also clever hacks; this would allow the apprentice both to catch blunders and to capture the rationale for unusual programming choices.

Interactions with the Apprentice were also envisioned to be "natural", i.e., largely indistinguishable from those one would have with a colleague. They would, therefore, include natural language, both spoken and written, and the use of diagrams and other informal drawings.

In retrospect, the idea was at least as wildly optimistic as it was visionary. Making the vision a reality required meeting several still daunting technical challenges:

- First, we believed that most large software systems are built from routine combinations of standard patterns and clichés. But the challenge is finding a representation that makes those commonalities obvious while overcoming the variations in surface syntax (due to programmer style or choice of programming language) that obscure the commonalities.
- The second challenge is to pre-analyze these clichés and patterns so that we know *a priori* how, why and under what conditions they work.
- The third challenge is to recognize instances of standard clichés and patterns within source code, making it possible to reuse prior analyses stored in the apprentice's knowledge base.
- The final challenge is to construct an apprentice system that would be as natural to interact with as a junior colleague. We need to build a system that can understand the informal diagrams and natural language that programmers and designers use when talking to one another.

Despite these challenges, the original Programmer's Apprentice project accomplished a great deal:

- It demonstrated that routine software is often realized as compositions of commonly occurring clichés and patterns (Rich 1987).
- It demonstrated that these clichés are best characterized not in terms of source code templates but rather in terms of the "plan calculus"—a formal language whose building blocks are control-flow and data-flow links that connect a collection of components specified by formal pre- and post-conditions and other invariants. In addition, a technique called temporal abstraction allows one to abstract away loops and recursions and instead to view the program as a pipeline of streaming components. This both

makes many commonalities more obvious and facilitates formal analysis (Waters 1979; Rich 1984; Shrobe 1979; Waters 1988; 1991; 1978).

- Finally, the original project demonstrated that it is possible to recognize instances of standard clichés and patterns within source code, making it possible to understand how and why a module works, by combining the analyses of the clichés (Rich and Wills 1990; Wills 1992; 1990; Clayton, Rugaber, and Wills 1998; Wills 1996).

Looking forward

Despite these intellectual successes, the Apprentice project didn't achieve its vision of providing a practical, knowledge-rich programmer's assistant. But there is reason to believe that it's time to revisit the concept, primarily because the technologies we can build on today are much more mature than were those of the 1970's:

- Computers are roughly 50,000 times more powerful on every relevant dimension than in the early 1970's. This has enabled techniques to be used in practice today that were unthinkably slow at that time.
- Our technologies for formal analysis of software are reaching practicality. Theorem proving and other formal methods have improved to the point where we can construct formal proofs of the correctness of operating system micro-kernels. Furthermore, many of today's advanced theorem proving systems (e.g., ACL2, Coq) include programming languages for expressing theorem proving "tactics". This allows us today to associate with every programming cliché a set of tactics for deriving proofs of the desirable properties of that cliché.
- It's much easier to build the kinds of symmetric, multimodal interfaces that facilitate natural interactions between programmers and an apprentice system. We have systems that routinely understand natural language (Katz et al. 2007; Katz, Borchardt, and Felshin 2006; 2005; Katz 1997) and sketches (Alvarado, Oltmans, and Davis 2002; Alvarado 2000; Alvarado and Davis 2006; Alvarado 2007; Hammond and Davis 2006; Sezgin, Stahovich, and Davis 2006; Adler 2003; Hammond et al. 2002; Stahovich, Davis, and Shrobe 1998; 1995).
- A large subset of all the software in the world is now available online in repositories that include source and object code, version histories, test cases, build scripts, etc. Vast stores of software are available in online open repositories. To first order, if you need a routine to accomplish a specific task, then, with high probability, there is a piece of software somewhere in one of those repositories that can do the job. If you want an example of something, it's probably out there somewhere. If you have just made a mistake in your code, it's likely that someone else has made a similar mistake, has checked in a fix to that mistake, and both versions are available in a version control system. If you are trying to prove a property about your system, it is increasingly likely that similar properties have been proven about similar systems.
- Machine Learning technology has greatly advanced, making it easier, in some cases, to acquire knowledge through data mining. It is plausible that machine learning techniques can be applied to software repositories to capture common patterns, thereby automatically populating at least some of the knowledge base of the apprentice.

Finally, the nature of the programming activity itself has changed enormously, making the availability of an apprentice-like system much more valuable:

- Today's software developers spend much of their time scouring open source repositories and interacting with experts via the internet (e.g., stackoverflow.com). But how much does that help us? Can we find what we need quickly? From what's available, can we tell whether some specific instance is going to meet our needs or even whether it's correct? Usually not.
- Programmers now work mainly in teams, and the average life of a programmer on a project is quite short. Preserving the reasons for design decisions and other forms of rationale is now a critical, not an optional, task.
- Much of programming today involves gluing together components of large frameworks rather than coding from scratch. Understanding the protocols for using these frameworks is necessary for using them appropriately. The rules for using a protocol are often best illustrated through examples and counter-examples. But finding such examples is often difficult, largely due to the low level of semantics employed in searching. Analyzing the examples in a repository and representing them in a semantically richer manner, such as that used in the Programmer's Apprentice, might make the task easier.
- Building and maintaining a complex, long-lived application involves a large number of tools and specialties, including: system design, functional requirements, security requirements, web frameworks, object–database mapping, software (update) delivery, license management, integration, testing, and verification. Much of the data management involved in sustaining such activities might well be better accomplished by an intelligent assistant.

All this highlights the pressing need to capture semantically rich annotations to software systems, and the need to maintain relationships between all the artifacts associated with a software system over time. This in turn raises many new questions and challenges:

- How do we formally represent and organize high-level design artifacts? How do these relate to both requirements and implementation artifacts? How have these designs changed over time?
- How do we manage the evidence that a particular implementation correctly implements all of (and only) the requirements and policies for the application? Are there formal proofs of correctness, and if so, how do we represent them in a way that facilitates reuse? Are there test suites, and if so, what is the coverage, of both requirements and source code?
- What is the end-to-end certification case for the correctness of the software system?
- How can all of these artifacts and rationale be maintained over time? As requirements change, or source code is

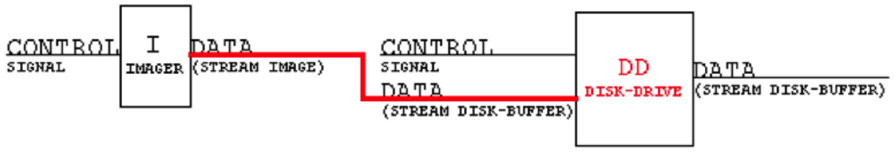


Figure 1: The Apprentice Catches A Blunder

updated, or even the tool-chain is updated, how is the certification case re-checked?

- Are there idioms, clichés, and patterns at all of these levels, not just at the implementation level? For example, constructing formal proofs has evolved to be more like software development. Are there proof clichés and patterns, and what is the relationship between the patterns and clichés used in a piece of software and those used in its proof?
- Can we make interaction with our design and implementation tools so natural that valuable information that is today routinely lost would instead be captured and made available?

A Scenario

As a step toward understanding how we might proceed in today's world, we have begun construction of a prototype apprentice system and have implemented enough functionality to conduct a simple design exercise with a user. In ths section, we will describe a part of that interaction, showing the system as it currently performs.

The current prototype interacts with its user using spoken language and a simple GUI interface for drawing. Spoken language is captured using the SIRI programmatic interface (on a MacBook) and is then parsed using an HTTP-based interface to the START natural language system (Katz, Borchardt, and Felshin 2006; 2005; Katz 1997). START returns the parse as a set of subject–relation–object triples. Similarly, to generate spoken output, the system sends a set of triples to START, asks it to produce text and then passes the generated text to the built-in text-to-speech system.

The prototype also contains a modest-sized ontology. It knows, for example, that:

- There are data-structures and computations.
- Sets, sequences, and mappings are types of collections, and collections are data structures.
- A sequence can be either a data structure (distributed in space) or a stream of values (distributed in time).
- An array can be viewed as either a sequence of rows or as a sequence of columns, and rows and columns can be viewed as vectors.
- There are standard clichés for iterating over sequences of values and for accumulating values into a container.
- There are input and output devices that produce and consume data of specific types.

Our prototype's design session deals with the construction of a software system that manages a sensor carried on an airborne vehicle. The initial design goal is to capture and store the outputs of the sensor.

The user begins by telling the Apprentice that there is an "imager" component of the system. The dialogue that ensues is as follows:

User: (selecting a box in the drawing) This thing is an imager.
Apprentice: I don't know what an imager is; can you tell me about imagers?
User: Imagers are input devices.
Apprentice: What type of output does an imager produce?
User: Images.

At this point the apprentice updates its ontology to include an entry for "imager", situating that entry under "input device". The apprentice asks about the type of the output produced by an imager because it already knows that any input device produces an output, but it has no further information about the type of output produced by input devices in general. The apprentice updates its entry for imagers, noting the type restriction on the output of imagers. It doesn't query any further because it already knows what an image is: it's an array of pixels. It also already knows that pixels can be viewed as a sequence of bytes.

Next the user adds a disk drive to the drawing and draws an arc connecting the output of the imager to the input of the disk drive. At this point the apprentice interrupts the user, pointing out that this is a mistake:

Apprentice: You can't connect the output of the imager to the input of the disk drive. Imagers produce images, which are arrays of pixels, but disk drives consume buffers which are vectors of bytes. (See Figure 1.)
User: Insert a transducer.

The apprentice updates the drawing by adding a new box, labeled "transducer" and redraws the arcs so that the output of the imager goes into the transducer while the output of the transducer goes into the disk drive. The drawing is also updated to show the required types of all the data flowing through the system. (See Figure 2.)

Next the user asks the apprentice to synthesize the code for the transducer. The apprentice doesn't actually know how to do this task directly, but it is able to use its reasoning capabilities as follows:

- The goal is to transform an image (an array of pixels) into a disk buffer (a vector of bytes). One way to do this is to employ a standard "generate and accumulate" cliché.
- The prerequisite for using this cliché is that the type of the output produced by the generator be a subtype of the type of the accumulator's input.
- Pixels aren't a subtype of bytes, but a pixel can be viewed

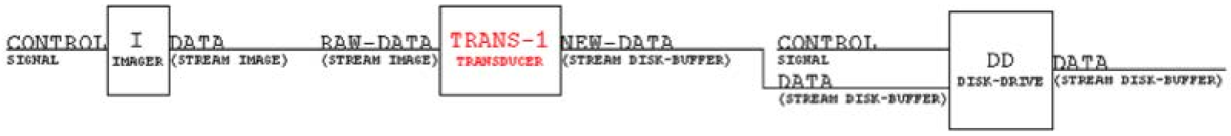


Figure 2: The User Asks the Apprentice to Insert a Transducer

as a vector of bytes (in either "big-endian" or "little-endian" order). So a second generator is inserted to transform the stream of pixels into a stream of bytes.

- One way to do that is using a standard generator cliché for linear structures. (The fact that vectors are linear structures is already in the apprentice's ontology.)
- Unfortunately, the apprentice also doesn't know how to generate the elements of an array. However, it does know that an array can be viewed either as a vector of rows or as a vector of columns. Furthermore, columns and rows can both be viewed as vectors. The apprentice, therefore, refines the design of the array generator into a cascade of two more instances of the cliché for generating linear structures; the first generates the rows (or columns) and the second generates the elements of each row (or column).

However, the apprentice has no reason for choosing one order of traversing the array over the other. It therefore queries the user:

Apprentice: I'm trying to generate the elements of the array. I can do this in row-major or I can do this in column-major order. Which do you prefer?

User: Row-major.

Apprentice: What is your reason?

User: Row-major is more efficient. Rows are stored contiguously. So traversing in row-major order gives greater cache locality.

The apprentice records both the particular choice made as well as the design rationale for it (i.e., it's more efficient). It also learns that there is usually a reason for preferring row-major order and this fact is added to its knowledge base. Similarly, there is a choice to be made as to whether to generate the bytes of each pixel in "big-endian" or "little-endian" order. The programmer chooses "little-endian" since this is the standard convention for information either in transit or in persistent storage. Again the choice and its rationale are captured and the background knowledge behind the rationale is added to the apprentice's knowledge base.

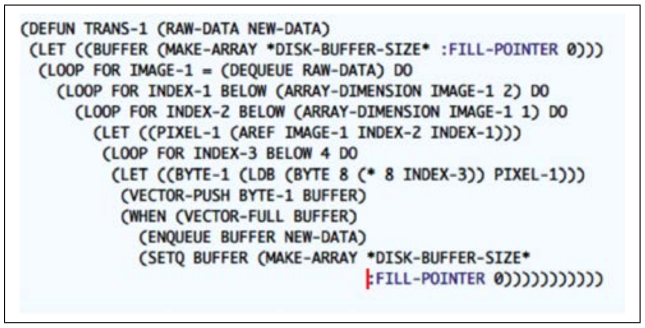
At this point the apprentice can transform its internal representation into the surface syntax of a programming language. We generate Common Lisp both because of the simplicity of its syntax and because of the high-level abstractions it affords. The transformation process is based on a variant of symbolic evaluation. The synthesized code is shown in Figure 3.

Summary

In this simple scenario we illustrated our initial efforts on the following points:

- The apprentice can interact in a natural manner using language and sketches of a sort similar to that of a human.
- The apprentice can learn through interactions with the user.
- The apprentice can discover and capture design rationale.
- The apprentice can catch simple blunders made by the user.
- The apprentice can take on routine coding tasks.

Earlier we noted that there a large number of other tasks that we might like an apprentice to undertake for us. Perhaps attacking all these issues would be as much of a "blue sky" effort today as was the original Programmer's Apprentice project. But it is one that brings together diverse areas of Artificial Intelligence in the service of an important need.



```
(DEFUN TRANS-1 (RAW-DATA NEW-DATA)
 (LET ((BUFFER (MAKE-ARRAY *DISK-BUFFER-SIZE* :FILL-POINTER 0)))
  (LOOP FOR IMAGE-1 = (DEQUEUE RAW-DATA) DO
    (LOOP FOR INDEX-1 BELOW (ARRAY-DIMENSION IMAGE-1 2) DO
      (LOOP FOR INDEX-2 BELOW (ARRAY-DIMENSION IMAGE-1 1) DO
        (LET ((PIXEL-1 (AREF IMAGE-1 INDEX-2 INDEX-1)))
         (LOOP FOR INDEX-3 BELOW 4 DO
          (LET ((BYTE-1 (LDB (BYTE 8 (* 8 INDEX-3)) PIXEL-1)))
           (VECTOR-PUSH BYTE-1 BUFFER)
           (WHEN (VECTOR-FULL BUFFER)
             (ENQUEUE BUFFER NEW-DATA)
             (SETQ BUFFER (MAKE-ARRAY *DISK-BUFFER-SIZE*
                                      :FILL-POINTER 0)))))))))))
```

Figure 3: The Generated Code

Acknowledgements

This material is based upon work partially supported by the Center for Brains, Minds and Machines (CBMM), funded by NSF STC award CCF-1231216, and partially supported by the Air Force Research Laboratory, under agreement number FA8750-11-2-0225. A version of this report appeared in the Proceedings of the AAAI Twenty-Ninth Conference on Artificial Intelligence (AAAI-15).

References

Adler, A. D. 2003. *Segmentation and alignment of speech and sketching in a design environment*. Ph.D. Dissertation, Massachusetts Institute of Technology.

Alvarado, C., and Davis, R. 2006. Dynamically constructed Bayes nets for multi-domain sketch understanding. In *ACM SIGGRAPH 2006 Courses*, 32. ACM.

Alvarado, C.; Oltmans, M.; and Davis, R. 2002. A framework for multi-domain sketch recognition. In *AAAI Spring Symposium on Sketch Understanding*, 1–8.

Alvarado, C. J. 2000. *A natural sketching environment: Bringing the computer into early stages of mechanical design*. Ph.D. Dissertation, Massachusetts Institute of Technology.

Alvarado, C. 2007. *Multi-domain sketch understanding*. ACM.

Clayton, R.; Rugaber, S.; and Wills, L. 1998. On the knowledge required to understand a program. In *Proceedings of the Fifth Working Conference on Reverse Engineering*, 69–78. IEEE.

Hammond, T., and Davis, R. 2006. Tahuti: A geometrical sketch recognition system for UML class diagrams. In *ACM SIGGRAPH 2006 Courses*, 25. ACM.

Hammond, T.; Gajos, K.; Davis, R.; and Shrobe, H. 2002. An agent-based system for capturing and indexing software design meetings. In *Proceedings of International Workshop on Agents In Design, WAID*, volume 2, 203–218.

Katz, B.; Borchardt, G.; Felshin, S.; and Mora, F. 2007. Harnessing language in mobile environments. In *Proceedings of the First IEEE International Conference on Semantic Computing (ICSC 2007)*, 421–428.

Katz, B.; Borchardt, G.; and Felshin, S. 2005. Syntactic and semantic decomposition strategies for question answering from multiple resources. In *Proceedings of the AAAI 2005 Workshop on Inference for Textual Question Answering*, 35–41.

Katz, B.; Borchardt, G.; and Felshin, S. 2006. Natural language annotations for question answering. In *Proceedings of the 19th International FLAIRS Conference (FLAIRS 2006)*.

Katz, B. 1997. Annotating the world wide web using natural language. In *Proceedings of the 5th RIAO Conference on Computer Assisted Information Searching on the Internet*, 136–159.

Rich, C., and Shrobe, H. E. 1976. Initial report on a LISP programmer's apprentice. Technical Report TR–354, MIT AI Lab. `http://hdl.handle.net/1721.1/6920`.

Rich, C., and Shrobe, H. E. 1978. Initial report on a LISP programmer's apprentice. *Software Engineering, IEEE Transactions on* (6):456–467.

Rich, C., and Waters, R. C. 1981. Computer aided evolutionary design for software engineering. *ACM SIGART Bulletin* (76):14–15.

Rich, C., and Waters, R. C. 1988. The programmer's apprentice: A research overview. *Computer* 21(11):10–25.

Rich, C., and Waters, R. C. 1990. The programmer's apprentice.

Rich, C., and Wills, L. M. 1990. Recognizing a program's design: A graph-parsing approach. *Software, IEEE* 7(1):82–89.

Rich, C.; Shrobe, H. E.; Waters, R. C.; Sussman, G. J.; and Hewitt, C. 1978. Programming viewed as an engineering activity. Technical Report AIM–459, MIT Artificial Intelligence Laboratory. `http://hdl.handle.net/1721.1/6285`.

Rich, C.; Shrobe, H. E.; and Waters, R. 1979a. Computer aided evolutionary design for software engineering. Technical Report AIM–506. `http://hdl.handle.net/1721.1/6314`.

Rich, C.; Shrobe, H. E.; and Waters, R. C. 1979b. Overview of the programmer's apprentice. In *Proceedings of the 6th International Joint Conference on Artificial Intelligence, Volume 2*, 827–828. Morgan Kaufmann Publishers Inc.

Rich, C. 1984. A formal representation for plans in the programmers apprentice. In *On Conceptual Modelling*. Springer. 239–273.

Rich, C. 1987. Inspection methods in programming: Clichés and plans. Technical Report AIM–1005, MIT AI Laboratory. `http://hdl.handle.net/1721.1/6056`.

Sezgin, T. M.; Stahovich, T.; and Davis, R. 2006. Sketch based interfaces: Early processing for sketch understanding. In *ACM SIGGRAPH 2006 Courses*, 22. ACM.

Shrobe, H. E. 1979. Dependency directed reasoning in the analysis of programs that modify complex data structures. Technical Report TR–503, MIT AI Lab. `http://hdl.handle.net/1721.1/6890`.

Stahovich, T. F.; Davis, R.; and Shrobe, H. 1995. Turning sketches into working geometry. In *ASME Design Theory and Methodology*, volume 1, 603–611.

Stahovich, T. F.; Davis, R.; and Shrobe, H. 1998. Generating multiple new designs from a sketch. *Artificial Intelligence* 104(1):211–264.

Waters, R. C. 1978. Automatic analysis of the logical structure of programs. Technical Report TR–492, MIT AI Laboratory. `http://hdl.handle.net/1721.1/6929`.

Waters, R. C. 1979. A method for analyzing loop programs. *IEEE Transactions on Software Engineering* 237–247.

Waters, R. C. 1988. Program translation via abstraction and reimplementation. *Software Engineering, IEEE Transactions on* 14(8):1207–1228.

Waters, R. C. 1991. Automatic transformation of series expressions into loops. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13(1):52–98.

Wills, L. M. 1990. Automated program recognition: A feasibility demonstration. *Artificial Intelligence* 45(1):113–171.

Wills, L. M. 1992. Automated program recognition by graph parsing. Technical report, MIT AI Laboratory.

Wills, L. M. 1996. Using attributed flow graph parsing to recognize clichés in programs. In *Graph Grammars and Their Application to Computer Science*, 170–184. Springer.