Just-in-Time Programming

Chapter
88

by
Richard Potter
Human Computer Interaction Lab
University of Maryland

Introduction

Many of the other chapters have presented advancements in programming by demonstration by presenting PBD systems and their innovations. In other words, these chapters have presented *solutions*. This chapter takes another tact by discussing PBD in the context of a *problem*. The problem is to create a programming system that is effective for just-in-time programming. This chapter defines just-in-time programming, explains how it relates to other forms of programming, and explores how creating effective just-in-time programming systems motivates PBD research.

Just-in-time programming is the implementing of algorithms during task-time (i.e. the time when the user is actually trying to accomplish the task to be automated) and can be characterized by a *situation* with the following components:

• a computer user who could be either a novice user or an experienced programmer,
• a task that the user is manually accomplishing and completion of which is the user's primary goal,
• an algorithm that will accomplish a subtask[1] (i.e. *part* of the task) and that the user envisioned while working on the task,
• and an attempt by the user to implement the algorithm for the purpose of more effectively completing the task.

In short, the goal of just-in-time programming is to allow users to profit from their task-time algorithmic insights by programming. Instead of automating with software that was carefully designed and implemented much earlier, the user recognizes an algorithm and then creates the software to take advantage of it just before it is needed, hence implementing it just in time.

It is worth emphasizing that the user's task could be from any domain (e.g. graphic drawing, scientific visualization, word processing, etc.) and that the algorithm to be implemented originates with the user. Obviously, a user with more programming experience will be able to envision a more complex algorithm than a novice user. How the user comes up with the algorithm is not a concern. Also, no hint of a solution appears in the problem statement. Any programming system could conceivably be used for just-in-time programming, including C, PASCAL, keyboard macros, scripting languages, or PBD. PBD will probably be an important part of the more successful just-in-time programming systems, but the problem statement leaves open the possibility for other solutions.

Just-in-time programming research shares many of the motivations of other PBD research. Chief among these is that users often do repetitive or algorithmic subtasks that the computer could be doing. We call these subtasks *potential computer subtasks* and call these situations *opportunities for new beneficial automation*. Because automating can increase productivity and user satisfaction and at the same time reduce errors, one would expect the user to delegate potential computer subtasks to the computer. That users often do not take advantage

---

[1]The word *subtask* will be used throughout the chapter to emphasize the relationship between the potentially automatable subtask and the overlying task

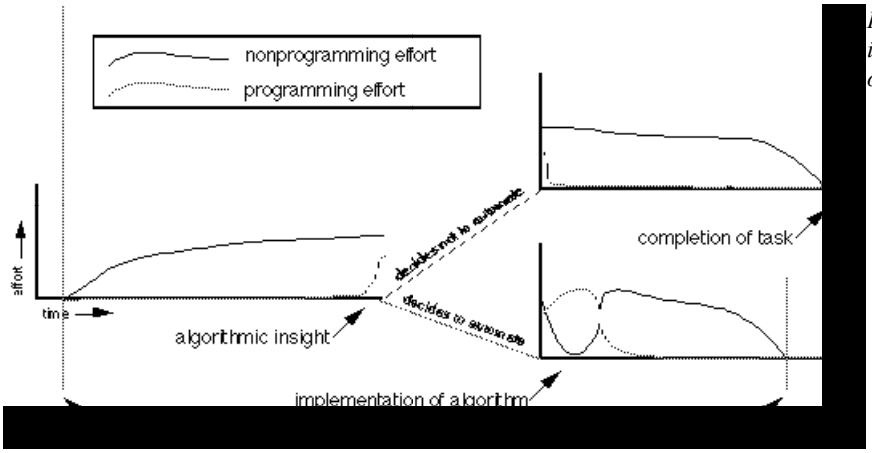

Figure 1: Just-in-time programming intermixes programming effort with other task related effort.

of these opportunities motivates researching ways to improve the computer. Just-in-time programming research and PBD research assert that easier to use *programming tools* will allow users to better take advantage of opportunities for new beneficial automation.

Just-in-time programming research, however, is focused on making programming easier for a specific cross section of situations. These situations are primarily defined by the user programming during task-time. In other words, the user is attempting to write a program for a task that is already in progress. Figure 1 summarizes the relationship between task progress and the user's expenditure of effort. The expenditure of effort for just-in-time programming is shown separate from the other task related effort. The difficulty of just-in-time programming results from the spreading of the user's mental resources between two activities.[2] Another difficulty is that the time spent programming contributes *directly* to total time between the start and completion of the task.

What is not Just-in-time Programming?

One might contrast just-in-time programming with, for lack of a better term, *task-time independent programming* which is summarized in figure 2. For an example, consider a user who is programming a HyperCard mock-up of

[2]See [Cypher 86] for a discussion on computer systems designed for multiple activities.

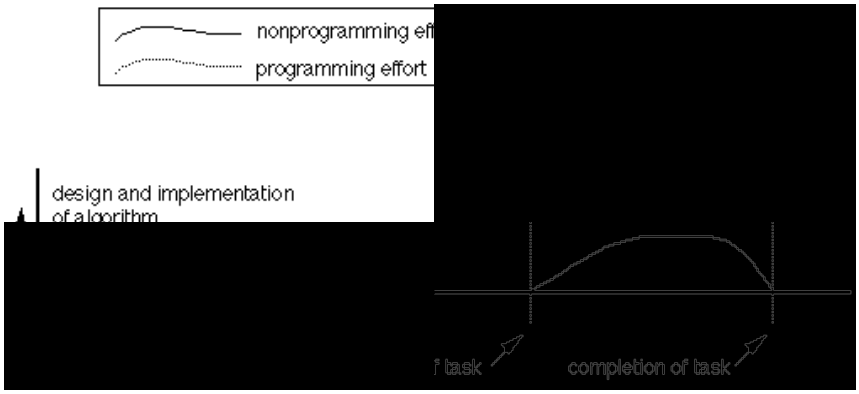

Figure 2: Task-time independent programming separates programming effort from other task-related effort.

a user interface in preparation for a demonstration the next day. The subtask of the software is to help an audience visualize a proposed user interface design during the meeting. This example would be task-time independent programming because the software is created well before the meeting. As is the case with many distinctions, there are examples that straddle the line between just-in-time programming and task-time independent programming, but the discussions that follow should hold regardless.

Just-in-time programming can be contrasted with other forms of programming by considering implications of the situation characterized in the introduction. For example, since the algorithm to be implemented is the product of the user's insight, it is typically simple. Thus one could contrast just-in-time programming with *programming-in-the-large*. For an example, developing a full featured word processor would not be just-in-time programming because an algorithm that implements all the features of a word processor is too complex for one person to envision.

Another implication of the situation characterized in the introduction is that the user has the option of avoiding programming altogether because the user can continue to manually accomplish the subtask. Thus one might contrast just-in-time programming with, for lack of a better term, *essential programming* where the necessity of programming is accepted. For example, a programmer who is creating software that performs a communication task on a satellite does not have the option of accomplishing the task manually. Thus the programming effort is essential.

How does just-in-time programming relate to the more common PBD application of *programming for novice programmers*? These two types of programming are basically independent; programming in a given situation could represent both, one, or neither of these. A novice programmer writing a short program that changes all numbers in a document to a larger font would be an example of both types of programming, assuming the user was about to make the modifications by hand. A novice programmer writing educational software to be used by students at a later time would be an example of a novice programmer programming, but not just-in-time programming. I recently modified a postscript file to only print out the even pages of the document and then the odd pages so that it would print on both sides of the paper without requiring me to issue a separate print command for each page. This would be an example of just-in-time programming that was not programming for the novice.

A Subtask Suitable for Just-in-Time Programming

I happened upon an opportunity for just-in-time programming when Think C updated their class library to version 1.1. Before version 1.1, rectangles were defined with 16-bit coordinates and in version 1.1 rectangles were defined by 32-bit coordinates. When I first compiled my software project with the new class library, type mismatch errors occurred where my software expected 16-bit values. Many of these were simple assignment statements. The new class library included a utility function for converting 32-bit rectangles to 16-bit rectangles, so a typical fix involved changing a line of the form `*inset=frame;` to the form `longToQDRect(&frame,inset);`. Various other types of errors were found and fixed as well. The second time an assignment of a 32-bit rectangle to a 16-bit rectangle caused an error, I recalled that there were many such assignments throughout my program and concluded I would, in time, be transforming many lines from assignment statements into function calls. Each would differ only in the names of the variables and whether each variable was a pointer or not (i.e. preceded by a "`*`"). For the rest of this paper, transforming one of these lines will be called the *line transformation subtask*.

To attempt to automate this subtask would have been just-in-time programming because of the situation. This particular example also contrasts well with the other types of programming discussed above. It would certainly not be task-

time independent programming because I was in the midst of modifying my software project. The subtask was algorithmically simple so it was not programming-in-the-large. Automating the subtask was not essential, so it was not essential programming.

So to break this situation down into the components of just-in-time programming:

The user:
myself

The task:
modifying a software project to work with an updated class library

The subtask:
changing certain lines of source code from the form {`*`}`var1 =` {`*`}`var2;` to the form `longToQDRect(`{`&`}`var2,`{`&`}`var1);` (i.e. the line transformation subtask)

The algorithm:
insert leading white space
insert "`longToQDRect(`"
if second variable name is not preceded by "`*`", insert "`&`"
insert second variable name
insert ","
if first variable name is not preceded by "`*`", insert "`&`"
insert first variable name
insert "`)`"
insert rest of line (the "`;`" and comments, if any)
delete original line.

The attempt to automate:
Actually I did not try to automate the subtask. The rest of the chapter will explain why.

Five Obstacles

One reason to explicitly state a problem is so that it can be broken down into meaningful subproblems. One way to do this is to analyze current technology, identify common obstacles that prevent the technology from being effective, and let the subproblems be to find ways to eliminate these obstacles. The following

```
void TransformLine(char *s)
{
 char var1[40], var2[40], rest_of_line[100];
 int  i,j,first_non_white;
 first_non_white = 0;
 while (isspace(s[first_non_white])) first_non_white++;
 i = first_non_white;
 j = 0;
 if (s[i]!='*') var1[j++] = '&';
 else i++;
 while ((!isspace(s[i])) && s[i]!='=') var1[j++] = s[i++];
 var1[j] = 0;
 while (isspace(s[i]) || s[i]=='=') i++;
 j = 0;
 if (s[i]!='*') var2[j++] = '&';
 else i++;
 while ((!isspace(s[i])) && s[i]!=';') var2[j++] = s[i++];
 var2[j] = 0;
 j = 0;
 while (s[i]!=0) rest_of_line[j++] = s[i++];
 rest_of_line[j] = 0;
 s[first_non_white] = 0;
 strcat(s,"longToQDRect(");
 strcat(s,var2);
 strcat(s,",");
 strcat(s,var1);
 strcat(s,")");
 strcat(s,rest_of_line);
}
```

Figure 3: A C algorithm that automates the line transformation subtask assuming the line has been isolated in a character buffer.

sections discuss five obstacles that often prevent users of current programming systems from profiting from their algorithmic insights. Each section also discusses PBD's potential role in solving the subproblem represented by each obstacle.

Effort of entering the algorithm

Given that I understood the algorithmic structure of the line transformation subtask, why not automate it? Since I was actually using a C programming system, let's first explore the possibility of using it. Figure 3 shows a program in C that can transform the line as needed, assuming that the line has been loaded into a string (we will deal with this assumption more later). One obstacle quickly becomes apparent: the *effort of entering the algorithm*. Merely the *physical effort* of typing in the 749 characters of this program would likely undermine the

benefits of automating this small part of the task.[3] But there is also the *mental effort* required to translate the algorithm into the idioms of the language and to work out the fine details. For example, after years of programming in C, I still must look up the **`strcat`** function in the manual to see if it copies from the first argument to the second or vice versa.

Discussion No matter what type of programming is being done, reducing this effort is desirable. However, the effort of entering an algorithm is particularly important for just-in-time programming because the subtasks appropriate for just-in-time programming are typically special purpose needs that can not be as widely applied as functionality that is more generic. The line transformation task is a good example of this because once I finished updating the software project, I had no need for this particular functionality. When the benefits per programming effort are modest, only minimal effort can be expended towards entering the algorithm before the venture becomes pointless. If the user's task is creative or involves problem solving, then the user can scarcely afford to expend mental effort for modest gains. Of course, there are times when the payoffs of just-in-time programming are large enough that the effort to enter the algorithm is not as crucial. But there are enough opportunities for modest payoffs that finding ways to reduce the effort required to enter the algorithm is an important subproblem to solve.

Sometimes creating new beneficial automation by programming pays off because the user can apply the automation many times in the future. A script that automatically dials a remote computer and logs the user into their account would be a good example. Here the distinction between just-in-time programming and task-time independent programming is blurred. Strictly speaking, the programming effort is expended at a time independent of when the benefits of the automation are received. However, the user is likely to automate this task at a time when they are about to dial in to the remote computer manually, that is, when the desirability of automating the subtask comes to mind. So although the user could set aside some time and do task-time independent programming, psychologically the task-time aspects of whatever the user is doing are likely to im-

---

[3]Shortening variable names and removing extra blanks can reduce this program to 438 characters.

pact the programming process. In this sense, much of the special concerns of just-in-time programming still hold.

Reducing this effort is less important for programming-in-the-large because the effort required to manage the complexity of a large software project tends to overshadow the effort required to enter the algorithm. In fact, programming languages such as Ada even increase the effort of entering the algorithm by requiring extra notation for modularizing the code. Such notation does not contribute directly to functionality, but is appropriate for programming-in-the-large because complexity is such an overriding concern.

Just-in-time programming accentuates the mental effort required to enter the algorithm because users must switch their mind-sets from the task domain to the programming domain as illustrated in figure 1. After the algorithm has been implemented, users must also expend effort to return their mind-sets to the task domain. Users might expend effort trying not to divert too much attention to the programming effort, sometimes trying to keep more in their short term memory than is reasonable. For creative tasks, this diversion is especially costly. In contrast, users who are programming independently of task time can change their mind-set over a longer period of time.

Therefore when minimizing the effort of entering the algorithm, it is important to minimize distraction from the task. One of the main sources of effort and distraction is the number of special programming concepts. For example, writing the C algorithm in figure 3 required remembering how strings are allocated and referenced. On this point, just-in-time programming and programming for novice programmers share similar goals because a programming system for novice programmers should require the understanding of as few new concepts as possible. In cases where a just-in-time programming system is being designed specifically for novice programmers, the same would apply.

The idea of just-in-time programming, however, is not limited to novice programmers. For expert programmers, whether a concept is familiar to a nonprogrammer is not the crucial factor. Instead, the programming system should require the user to understand only concepts that can be *ingrained* and that the user can apply *fluently*. Therefore, part of the research agenda of just-in-time programming should be to identify key skills that, if ingrained, will allow a user to more effectively write programs just in time. These skills could be anything

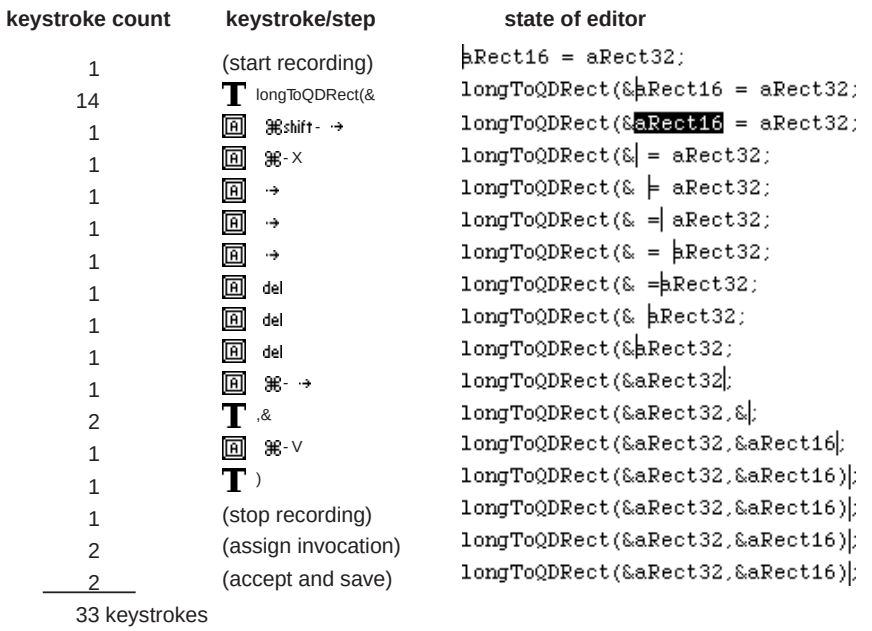

Figure 4: This Quickeys macro can partially automate the line transformation subtask. The number of keystrokes required to enter the macro and the visual state of the editor are shown.

from something classic like regular expressions to some new esoteric programming paradigm. It seems clear that, at least for the foreseeable future, users will have to understand the basic concepts of conditionals and iteration.

Solution directions Many techniques including code templates, code reuse, domain specific functionality, subroutines, copy/paste, and on-line documentation can help reduce the effort required to enter an algorithm. PBD helps reduce this effort by allowing users to enter the algorithm using the same interface as they would normally use to work the subtask manually. This helps reduce both the physical effort and the mental effort because the user is often well practiced at using this interface. Since the user would use the same user interface to work the subtask manually, the artifacts are already in short term memory and programming with them is likely to be less distracting than with an off-line programming language. The effort to enter the algorithm is also reduced because user interfaces are usually optimized to the task.

For a simple example of how PBD can reduce the effort of entering an algorithm, consider one partial solution to the line transformation subtask. If the user first places the cursor to the left of the first variable in the line to be trans-

formed and neither variable is a pointer, then the Quickeys macro shown in figure 4 will transform the line as required. The macro also assumes that exactly three characters (" = ") separate the two variable names. Only the 33 keypresses shown in figure 4 are required to implement the macro.[4] The visual feedback of the editor also helps reduce the mental effort by showing intermediate results.

Limited computational generality

Why illustrate the virtues of the keyboard macro by only partially automating the line transformation subtask? The reason is that the subtask requires conditional logic to decide whether each variable is a pointer or not. Keyboard macros only record straight-line algorithms and thus are not able to fully automate this subtask.[5] This illustrates an obstacle that users face when programming just in time: the programming systems that make it easy to enter their algorithm can often only implement algorithms of limited computational generality.

Discussion It is important for a just-in-time programming system to have full Turing-complete computational generality because there is no way to predict which of the vast array of algorithms the user might envision. Unfortunately computational generality is not one of PBD's strengths. Halbert recognized this when implementing SmallStar and concluded that control structures were better created by editing a static representation of the program than by demonstration [Halbert 84]. Others have used inference to generalize straight-line demonstra-

---

[4]The C algorithm in figure 3 can only be shortened to 623 characters by making the same assumptions as this macro.

[5]This is true of keyboard macros in general. Quickeys has an extension facility so there is a possibility that someone has written an extension that allows it to fully automate this subtask. Interestingly, it is actually possible to automate this subtask in the Emacs text editor using only straight line macros by leaning heavily on Emacs' underlying functionality. The trick is to narrow the editing region down to the current line and perform a series of clever search and replaces. The solution works but thinking of it takes some effort. Conditional logic would be far more straight forward.

tions into procedures with control structures. Cypher's Eager and Myers' Peridot used domain knowledge to infer procedures with control structures solely from straight-line demonstrations [EAGER CHAPTER, PERIDOT CHAPTER]. The computational generality of these systems, however, was limited by limited domain knowledge.

Solution directions In order for a PBD to be used for just-in-time programming, it will have to be integrated with other techniques to give full computational generality. Interesting directions include giving separate examples for each path of the algorithm as in Tinker [TINKER CHAPTER], or a combination of multiple demonstrations, inferencing, and special instructions from the user as in Metamouse [METAMOUSE CHAPTER].

Effort of invoking algorithm

As stated previously, the effort to enter an algorithm is less of an obstacle when the benefits of automating are large. So for the sake of argument, assume that I knew there would be hundreds of lines needing to be transformed and decided to automate the task using C. Limited computational generality would not be an obstacle with C. Are there other obstacles?

When the compiler detected an error in my software project, it would load the file containing the error into its text editor and highlight the erroneous line. To take advantage of the line transformation program, I would first have to judge if it was one of the simple type mismatch errors that could be fixed by the simple line transformation. If so, I would then invoke the C implementation on the specific line. But how would I do that? One possibility would be to mark the line some special way, perhaps by placing a "*****" at the beginning of the line, and then save the file out to disk. Then I could run the C program which would then prompt me for the name of the file with the incorrect line. The C program would then scan through the file for a line that started with a "*****" and apply the transformation to it. But this would be silly. The *effort to invoke the algorithm* would undermine the benefits and would be yet another obstacle to automating this subtask.

Discussion As in this case, a subtask appropriate for just-in-time programming typically applies to part of a larger document. Thus, users must be able to implement the algorithm such that they can specify which part of the document should be processed when they invoke the algorithm. It is important that they

be able to do this with ease because the benefits to be obtained by automating can be easily negated by the invocation effort. Unlike the effort to enter the algorithm, the effort to invoke the algorithm can not be amortized over the life of the new beneficial automation. The effort must be small in comparison with the benefit received from *each invocation* of the algorithm. When the payoffs per invocation are larger this obstacle is not as crucial, but enough opportunities for modest benefits exist that it is important to reduce the effort required to invoke algorithms implemented just in time.

For an example of how crucial the ease of invocation can be towards making automation beneficial, consider the feature on many word processors that allows a user to select a word simply by double clicking on it. The word processor automatically does the tedious subtask of extending the selection out to the word boundaries. Identifying these word boundaries manually is a simple subtask, so not much benefit is received each time the feature is used. However, words are selected so commonly that, over time, the feature is very beneficial. Another invocation strategy could easily undermine this benefit. For example, even requiring the user to click on the word and then select the feature from a pull down menu could require too much effort.

As the previous example implies, this obstacle is not unique to just-in-time programming. Because the effort to invoke the algorithm can not be amortized, any programming endeavor that produces interactive software needs to pay special attention to this obstacle. The main difference for just-in-time programming is that the user can not amortize the effort to *create* the invocation scheme as much.

Solution directions How should just-in-time programming make it easy to invoke algorithms? One clue is strongly implied by the hypothetical consideration of C for automating the line transformation subtask: users should be able to implement their algorithms such that they can perform the subtask without having to save their documents to disk. Instead their algorithms should be able to process data in its present form which is usually internal to some application. Thus just-in-time programming systems should allow users to process data within their applications. In addition, users should be able to use the application's data selection mechanisms to indicate what part of their document to process. This would enable users to work manually, apply a newly implemented al-

gorithm to the data, and continue to work manually without the overhead of saving the data to a file. Processing data within applications is central to PBD, so it already goes far to easing the effort to invoke the algorithm.

Beyond automating within applications, just-in-time programming systems should allow the user to choose among various invocation strategies. Standard invocations such as menu selections and keypresses should be supported. The ability to create more refined invocations, like double clicking on a an object to apply some automation to it, would be important for making some highly interactive automation worth creating. PBD techniques could possibly be used to have the fact that the user has started doing the subtask be what triggers the automation to be invoked. David Maulsby's Turvy and Metamouse give hints of how this might work [TURVY CHAPTER, METAMOUSE CHAPTER].

Inaccessible data and operators

So far we have seen several reasons to want to process data while it resides within an application. One is to make algorithms easier to enter by allowing the user to demonstration the algorithm through the user interface of the application. Another is that invocations can be made easier if the data is processed within the application. The Quickeys solution in figure 4 had these advantages, but it only partially automated the subtask. No other programming system on my computer (including APL, C, Lisp, Scheme, or HyperTalk) can automate within Think C's editor because of the fourth obstacle, *inaccessible data and operators*. In this case, this obstacle undermines the modest benefits of automating the subtask. If the benefits per invocation were greater, then accessing the data independently of the application by saving the document to a file might have made creating the new automation worthwhile.

In other cases, limited data access can take the simplest algorithms and render them impossible to implement. Consider the example discussed in [TRIGGERS CHAPTER REFERENCE] of automating the wrapping of a text field with a properly sized rounded rectangle. The algorithm to automate this task is trivial when stated in terms of the text field's and the rounded rectangle's properties of location, length and width. The central part of the algorithm is to set the rounded rectangle's location a bit above and to the left of the text field's location, and set the rounded rectangle's length and width to be a bit larger. Automating this task independently of MacDraw II would involve extracting

these properties from MacDraw II's coded file format, which would be very difficult. Also, the user's algorithm may be based on special functionality provided by the application such as, in this case, the ability to create rounded rectangles. Just-in-time programming systems should therefore be able to access properties from applications and invoke the operators provided by applications.

Sometimes an opportunity for new beneficial automation involves not so much the processing of data, but rather the repetitive manipulation of an application's user interface artifacts. For example, the user may wish to automate the toggling between two window arrangements. The only way a programming system can automate this is to access the state of the user interface and manipulate its components.

Discussion Inaccessible data and operators is a particularly common obstacle for just-in-time programming because users must make do with whatever form their data is in when they envision the algorithm. Usually this data exists within an application. In contrast, task-time independent programming often allows the user to plan what form the data will be in when the automation is eventually used. Many programming efforts, like games or educational software, are closed systems where the programmer can choose the format of the data to be whatever makes their programming effort easiest.

Solution directions In order for a programming system to access the data and operators of an application, there must be a communication protocol that both the programming system and the application follow. One way to effect this protocol is to build the programming system into the application. This strategy, however, limits the data access to the one application, so inaccessible data and operators would still be an obstacle when the user's algorithm involved multiple applications. Therefore just-in-time programming systems should make use of specially established interapplication protocols like Apple Events [Apple 91].

Sometimes programming systems can overcome the inaccessible data and operators obstacle by using protocols established for reasons other than interapplication communication. For example, the Quickeys solution uses the computer's keyboard input stream as a protocol to process data in the editor application. The Triggers chapter discusses an extension to this technique where pixel data from the computer display can be used to gain a significant degree data access

from any application. PBD plays a large role in making this extension possible because the algorithms implemented using these techniques sometimes contain large bitmap constants. Entering these bitmaps would be unwieldy if they could not be specified by demonstration.

Risk

The fifth obstacle is the risk that the automation will fail, be ineffective, or produce unintended results. Consider the risks of automating the line transformation subtask. There are many possible scenarios. In the best case the algorithm could have been entered almost effortlessly, and as each occurrence of a line needing the simple transformation was flagged by the compiler, I could have easily invoked the algorithm somehow. To my surprise, perhaps more chances to use the new automation occurred than were anticipated, making the automation pay off more than expected.

But there are many other possible scenarios. The algorithm could have taken a long time to enter, perhaps because some special purpose function had to be looked up in a manual. A mistake in the implementation might have caused the new (not beneficial) automation to destroy part of the source file, perhaps too quickly to be noticed. Limited data access could have turned the simple algorithm into one that was impossible to implement. I was not sure exactly how many more assignments of 32-bit rectangles to 16-bit rectangles were left in my software project, and thus there may have been too few to make the programming effort worthwhile. Unforeseen special cases may have made the envisioned algorithm simply wrong.

A user who is considering a just-in-time programming effort has the option of continuing to work manually. Given the many adverse scenarios, it is not surprising that the user would choose this option. Thus just-in-time programming systems often fail *because the user chooses not to use it*.

Risk was the main reason I chose not to automate the line transformation subtask. The partial solution using keyboard macros was the only one worth considering because it was the only solution that did not require saving the file to disk. In the past, my attempts to use keyboard macros have often been thwarted by unforeseen special cases, the difficulty of accommodating special cases into an already existing macro, and the uncontrollable speed of macros that make it difficult to verify that the macro works correctly. In retrospect, a keyboard

macro would have been worthwhile and would have prevented a few recompiles caused by typos in my manual transforming of the lines. However at the time, the apparent risks convinced me to play it safe and transform the lines manually.

Discussion Certainly all programming involves risk. The risks of just-in-time programming are notable on two accounts. One is that the time and effort spent implementing the algorithm relate *directly* to the success of the venture. For example, any extra time or frustration involved in automating the line transformation subtask would have quickly eliminated the potential benefits. In contrast, task-time independent programming efforts often relate only indirectly to success. For example, say a user is programming an animated demo for a five-minute presentation. If the demo take two hours longer to implement than expected, the presentation the next day can still be a success.

But the main reason risk affects just-in-time programming so strongly is that it is easy for users to choose to continue to work manually and avoid the risk. In contrast, essential programming requires users to make the best of what their programming systems have to offer. Users still have to assume whatever risks are present, but the programming system will not fail for lack of use.

Risk is caused, in part, by the users' uncertainty about how the other four obstacles will affect their attempts to automate. Thus, one way to reduce the risk is to work towards eliminating these four obstacles. For example, if entering the algorithm were effortless, there would be no risk in taking that step. Unfortunately, entirely eliminating these obstacles is very unlikely. In addition, users will still have to assume the risk that their algorithms might not do as expected. Thus, it is important to explicitly consider techniques that reduce the risks of just-in-time programming.

Solution directions One approach to addressing risk is to make it so that the user can accurately judge the effort required to implement the algorithm and accurately judge the benefits. Simplicity and visibility are two attributes of a programming system that would contribute to this approach. When users can confidently judge the benefits will be greater then the efforts, then they can proceed to profit from using the just-in-time programming system and the system will not fail from lack of use. The limitation of this approach is that merely judging the risks is a risk in itself because the user must expend some mental effort. Because it is unlikely that this effort can be eliminated entirely, the user will

```
step 1-read in a list of the file names in the folder containing the chapters
step 2-remove any file name from the list that contains the string "backup"
step 3-while the list is not empty repeat steps 4-9
    step 4-have MacWrite II open the file with the name appearing first in the list
    step 5-invoke MacWrite II's "Save As..." feature:
        step 6-select Text Only option
        step 7-append "(text)" to the file name
    step 8-close the document
    step 9-remove first file name from the list
```

Figure 5: An algorithm that will convert a folder full of MacWrite II files to pure ASCII.

have good reason to simply continue working the task manually without ever giving the opportunity to use just-in-time programming a second thought. Other techniques need to be considered.

Another approach to reducing risk is to enable users to profit from partial implementations of their algorithms. This would help alleviate the risk that an obstacle might prevent the implementing of part of the algorithm, render the whole algorithm useless, and waste any effort already expended. Users should be able to implement and profit from parts of their algorithms without requiring the entire algorithm to work flawlessly.

For example, assume a user has 20 book chapters saved as a separate MacWrite II files in a folder, and that a colleague requests a pure ASCII copy of each chapter. To manually convert each chapter to pure ASCII, the user would have to load its file into the MacWrite II, select the **Text Only** option, and save it back out to disk using a different file name. Assume files with the word "backup" in the file name should not be converted. The simple algorithm in figure 5 could select all 20 chapters in turn and carry out these repetitive actions. If the user were able to implement this algorithm, the tedium of keeping track of which files have been converted and the tedium of the repetitive actions would be avoided.

If the entire algorithm could be implemented confidently, easily, and flawlessly, then risk would not be an issue. Unfortunately, any step in the algorithm could cause potential problems. What if after putting some effort into the algorithm, the user discovers that only a subset of the nine steps can be implemented? For example, what if step number 7 could not be implemented because the user's programming system could not invoke the operator that selects the **Text Only** operator? The partial implementation can still potentially be beneficial if the

user can manually do the steps that prove difficult to implement. For example, the algorithm could invoke the "Save As..." dialog box and pause while the user selects **Text Only** manually. The partial implementation would still be beneficial because it would take care of the tedium of keeping track of which files have been converted and a great majority of the other actions.

For step number 7, this strategy is easy to imagine because the application's existing user interface can allow the user to carry out the hard to implement action manually. But what if step number 2 is too difficult to implement because the user's programming system has no built-in test for substrings? In this case the list is in the programming system's execution environment, not the application. If the user is to manually accomplish this step, the programming system must have an existing user interface that allows the user to manipulate the execution environment. Some interactive programming systems and debuggers allow users to modify the execution environment during run-time, but few allow data to be manipulated easily enough for the user to do real work. Also, the more that control has to pass between the user and their implementation, the more essential it will be that flexible invocation schemes are possible.

Essentially the user's risk is that the manual method might be more effective than implementing the algorithm. Therefore, another approach to addressing risk is to allow the user to pursue both alternatives in parallel. In theory, the risk of attempting to automate the subtask would be eliminated because if unforeseen difficulties make the programming effort ineffective, then the user can fall back on the manual method already underway.

In practice, this approach would probably not eliminate risk, but it could reduce risk greatly. PBD could play a large part in realizing this approach because it allows the user to implement algorithms by demonstrating on their actual task data. In other words, the user can be programming and manually accomplishing the subtask simultaneously. For example, recording the keyboard in figure 4 actually transforms one of the lines, so progress towards completing the overall task is hindered minimally.

This technique has its greatest potential when mixed with history based techniques. For example, Allen Cypher's system Eager records the user's actions into an event history [EAGER CHAPTER]. When Eager detects the user doing repetitive actions, it indicates this to the user by highlighting what it expects the

user to select next. For certain classes of algorithms, the user can implement an algorithm at almost no risk because the user takes no special actions. The decision of whether to invoke the algorithm still involves some risk because the exact behavior of some algorithms is difficult to predict. Therefore additional techniques such as undo and slow motion execution will have to be extended and refined.

Conclusion

Is creating a programming system that is effective for just-in-time programming an interesting research problem? The previous sections clearly show that it has not been solved already, so it meets this criterion. A second criterion for interesting research is that there be some indication that solutions are possible. The previous section touched on several promising research directions, many of which are based on PBD. A third, important criterion for interesting research is that it lead to tangible benefits. The benefits of improved just-in-time programming systems would be to allow users to better automate repetitive subtasks that arise from their unique circumstances. The line transformation subtask was one such example where automation would have led to significant benefit. There will always be subtasks like this that slip through the prepackaged functionality of applications because they result from the interactions of users with the complexities of the real world. Task-time is often the only possible time to implement the algorithms that can automate these subtasks.

But is it necessary to focus the research problem on such a narrow slice of programming to make automating these subtasks practical? After all, many of the obstacles facing just-in-time programming also affect other types of programming; it is possible that researching other types of programming will produce effective just-in-time programming systems as a side effect. Are there reasons for researching just-in-time programming specifically?

One reason is that a user must accomplish all of the following during task-time: assess risk, enter the algorithm, design the invocation scheme, solve data access problems, invoke the algorithm, verify correct program behavior, and resume work on the overlying task. Thus it is crucial that techniques that support these activities be refined to a degree that other forms of research are unlikely to achieve. For example, research that concentrates on programming-in-the-large is unlikely to adequately reduce the effort required to input the algorithm when

managing program complexity is its overriding concern. Research that assumes essential programming is unlikely consider techniques that reduce risk by enabling productive use of partially debugged programs. Research that assumes task-time independent programming is unlikely to recognize that programming may not be the user's primary concern. Thus it is unlikely to give adequate emphasis to minimizing distractions from the user's primary task. It is also unlikely to motivate data and operator access that is flexible enough to process the user's data wherever it may be when the opportunity to apply just-in-time programming arises.

Another reason is there are solutions that are appropriate for just-in-time programming but are not necessarily appropriate for other types of programming. For example, the pixel based techniques of Triggers [TRIGGERS CHAPTER] would not be appropriate for software that must run in the background. The technique of programming and accomplishing the task at the same time discussed in the risk section does not make sense for programmers who are writing software for other people's use.

By recognizing the special nature of just-in-time programming and by addressing the limitations of current programming systems head-on, much research progress should result. The five obstacles provide a set of subproblems that can be used to focus multiple avenues of research. Researchers should be careful not to accentuate one obstacle in the elimination of another; it only takes one to prevent a just-in-time programming system from being effective. It is crucial that a just-in-time programming system address risk because it is probably impossible to create a programming system where the user's every attempt at creating new automation will be profitable. The goal should be to create a programming system where a user can know that in the worse case attempting just-in-time programming will not hinder progress towards completing the task. Then the user will be able to confidently use the full extent of the programming system to profit from their algorithmic insights.

References for Alan

[Apple 91] Apple Computer, Inc., <Inside Macintosh, Volume VI> Addison Wesley, Reading, Massachusetts, 1991.

[Cypher 86] Cypher A., "The Structure of Users' Activities," <User Centered Design>, Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1986.

[Halbert 84] Halbert D., "Programming by Example," Ph.D. Thesis, Department of Computer Science, University of California at Berkeley , 1984.