Deliberated Evolution:  
Stalking the View Matcher in Design Space  

John M. Carroll and Mary Beth Rosson

ABSTRACT  
Technology development in HCI can be interpreted as a co-evolution of tasks and artifacts. The tasks people actually engage in (successfully or problematically) and those they wish to engage in (or perhaps merely to imagine) define requirements for future technology, and specifically for new HCI artifacts. These artifacts, in turn, open up new possibilities for human tasks, new ways to do familiar things, entirely new kinds of things to do. In this paper we describe psychological design rationale as an approach to augmenting HCI technology development and to clarifying the sense in which HCI artifacts embody psychological theory. A psychological design rationale is an enumeration of the psychological claims embodied by an artifact for the situations in which it is used. As an example, we present our design work with the View Matcher, a Smalltalk programming environment for coordinating multiple views of an example application. In particular, we show how psychological design rationale was used to develop a view matcher for code reuse from prior design rationales for related programming tasks and environments.

1. TASKS AND ARTIFACTS  
In 1605, Sir Francis Bacon called for a “natural history of trades.” He urged that technical tools, techniques and processes be made more public and explicit. This was one element in his broader project of developing practical science, and hinged on the assumption that if such knowledge could be more systematically considered and integrated, human progress would necessarily result. Thus, Bacon suggested that new concepts and inventions would result “by a connexion and transferring of the observations of one Arte, to the use of another, when the experiences of several misteries shall fall under the consideration of one man’s minde.” (1970: Book 2, p. 10)

Half a century later, when the Royal Society of London and the French Académie des Sciences were established, Bacon’s proposal was high on their agendas. However, the Royal Society quietly but promptly abandoned the project. And the Académie pursued it rather haltingly over the next century, culminating in the publication of Diderot’s Encyclopédie ou dictionnaire raisonné des science, des artes, et des métiers. Though Diderot’s objectives were explicitly those Bacon had originally articulated, the final product of the Encyclopédie substantially lagged the leading edge of technology and its directive impacts on technological evolution were relatively minor.

In this paper we are disciples of Bacon in an approach to HCI that we call “deliberated evolution”: we want to integrate understanding systems and building systems within a single framework of research-development. Our focus is on the psychological import of HCI artifacts (workstation hardware, operating systems, application programs, user interface displays and devices, and so forth). We are exploring the directive role that psychological design rationales of HCI artifacts can play in design work. In Bacon’s terms, our interest is in the connection and transferring of one design to a use in another design by means of explicit design rationale and systematic design methods.

There are many reasons why Bacon’s original proposal failed to effectively guide seventeenth-century technology evolution (Ferguson, 1977). One factor is that it just took

1 Current address: Center for Human-Computer Interaction, 660 McBryde Hall, Virginia Tech, Blacksburg, VA 24061-0106 U.S.A
too long to develop the “natural history of trades.” Our case study stresses the utility of concurrently designing new technology and analyzing its scientific rationale. We discuss the View Matcher, a software tool originally designed to address learning problems inherent in the Smalltalk language-environment, and subsequently redesigned as a tool to facilitate code reuse by skilled Smalltalk programmers. The several cycles of View Matcher design exemplify the deliberated evolution of technology that we see as Baconian.

Most technical activity in HCI can be framed as transaction between tasks and artifacts. The tasks people actually engage in (successfully or problematically) and those they wish to engage in (or perhaps merely to imagine) define requirements for future technology, and specifically for new HCI artifacts. These artifacts, in turn, open up new possibilities for human tasks, new ways to do familiar things, entirely new kinds of things to do. They also create new complexities of learning and performance, new interactions among tasks, and of course, new errors and other difficulties for users. The new tasks eventually devolve into requirements for further technology evolution, provoking further transaction (Figure 1).^2

![Figure 1: The transaction between tasks and artifacts.](image)

Examples of this are pervasive in HCI, but particularly good ones inhere in the particularly momentous technological developments in the field. Consider the spreadsheet. The first electronic spreadsheet, VisiCalc, was brought to market in 1979. It clearly embodied a simple, yet powerful response to a set of extant tasks: table-based calculations. Placing the spreadsheet in an electronic medium permitted accurate calculation and convenient handling. But it did much more. It opened up important new possibilities for table-based calculation tasks. Electronic spreadsheets facilitated projective analyses (what-if reasoning about altered conditions and abductive reasoning from a desired result to conditions that could produce it). Users could easily alter values and recalculate. Indeed, spreadsheets even afforded a kind of ad hoc work integration: users could type a memo describing an analysis right into a spreadsheet cell (Mack & Nielsen, 1987).

This evolution of spreadsheet tasks can be viewed as successively altering requirements for spreadsheet systems. Thus, in the early 1980s Context MBA provided integrated support for windows, graphing, word processing, and file management, for

^2 Though most technology development is evolutionary (Basalla, 1988), some clearly is not. Following a suggestion by Randy Smith, we identify our concern here as "normal" technology development, on analogy with Kuhn's (1970) notion of "normal science."
example displaying the spreadsheet in one window and a graph of selected cells or a report
in another. Lotus 1-2-3 introduced natural order recalculation (in which cell dependencies
determine the order of recalculation), easing the overhead of what-if explorations. These
advances, in turn, can be seen as encouraging further task evolution. For many users, the
spreadsheet environment became a fulcrum for work: planning, communicating, accessing
information, reporting, and presenting. It is now typical for spreadsheet systems to support
multiple windows, to integrate support for text and graphics, to share data with other
programs. Many spreadsheets offer a range of recalculation options to facilitate projective
analysis; some offer a “solver” function that takes a specification of a desired result and
suggests how to obtain it.

If we take the transaction between tasks and artifacts seriously as the evident
framework of technology evolution in HCI, we can ask how this structure might be more
deliberately managed and directed. In the development of the spreadsheet, successive task-
artifact excursions were not, as far as we know, guided by a public and explicit “natural
history of the spreadsheet,” to borrow Bacon’s term. Thus, even the summary
understanding encapsulated above was developed after the main points were already
embodied in new spreadsheet designs (Licklider, 1989; Mack & Nielsen, 1987).
Nevertheless, the prior tacit — or at least private — understanding of spreadsheet tasks,
and attendant technological limitations and possibilities, worked pretty well in guiding the
evolution of spreadsheet applications.

Could a better Baconian tool could produce even better results? Our approach to this
has been to directly augment the transaction manifest in current practice with explicitly
managed tools and representational techniques. We want to enhance the natural ecology of
HCI technology evolution, but not too radically to distort or undermine what is in essence a
fairly successful framework for design work and practical science (and to which designers
are now committed through their practice). We are seeking streamlined techniques to
supplement this framework with additional deliberate analysis and record-keeping.

Figure 2: A task-artifact framework for design in HCI.

Figure 2 schematizes our augmented task-artifact framework for HCI research-
development. Briefly, we propose guiding the discovery and integration of design
requirements by means of a scenario-based methodology (e.g., Carroll & Rosson, 1990,
1991; see also Carroll, Thomas & Malhotra, 1979; Guindon, 1990; Wexelblat, 1987). We
propose capturing and projecting possibilities of use in a psychological design rationale,
developed by claims analysis (e.g., Bellamy & Carroll, 1991; Carroll & Kellogg, 1989). We further suggest that by collating and abstracting such design rationales across collections of user interaction scenarios and artifacts a pertinent psychology of user tasks can be constructed (Carroll, 1990b; Carroll, Kellogg & Rosson, 1991).

In essence, we are attempting to enhance design practice by prescribing an incrementally more systematic version of the very practice we now can observe. For example, user interaction scenarios are already widely used for envisioning systems before they are built (they supplement, and sometimes supplant textual specifications in many development groups). Scenarios are required for the design of task oriented training materials and documentation (Carroll, 1990a) and for the design of user testing instruments (Roberts & Moran, 1983). Similarly, developers need to place their work within a context of ideas and techniques to understand what they have accomplished and what they want to accomplish subsequently. Such efforts are already common and important in HCI, though often informal (e.g., Smith, Irby, Kimball, Verplank & Harslem, 1982).

In the balance of this paper, we develop an example to illustrate how our notions of scenario-based design, task psychology, but especially, in the context of this special issue, psychological design rationale can work together within the task-artifact framework for research and practice in HCI.

Figure 3: View Matcher window for the blackjack game application. The Stack View is in the upper left; the Application View is in the upper right; the Class Hierarchy View is in the lower right; the Commentary View is in the lower left; the Inspector View is above the Commentary View. Note that in the actual Smalltalk VPM implementation, views are distinguished by color and grey level.
2. A VIEW MATCHER FOR LEARNING SMALLTALK

The View Matcher, shown in Figure 3, is a structured browser for Smalltalk/V (Carroll, Singer, Bellamy & Alpert, 1990). It presents multiple views of a running Smalltalk application, for example, a blackjack game. The various views are derived from the major system tools in Smalltalk (the debugger, the class hierarchy browser, the inspector), but they are simultaneously opened and displayed with the application graphics, and are jointly updated whenever the user interacts either with the application or with one of the tools. The View Matcher is intended to coordinate a programmer’s developing understanding of Smalltalk and its environment. It has been incorporated into a minimalist curriculum for Smalltalk now in use in IBM (Rosson, Carroll & Bellamy, 1990).

The design of the View Matcher was based on our understanding of the possibilities for learning inherent in the Smalltalk environment. This understanding rests on our analysis of new users as active learners, opportunistically pursuing personally meaningful goals, and trying to make sense of their experiences (Carroll, 1990a). A learning process of this sort entails three characteristic user concerns: one in which the learner has no idea what goals to pursue and wonders “What can I do?,” a second in which the learner finds and investigates a system object, wondering “How does this work?,” and a third in which the learner has some concrete goal in mind and wonders “How do I do this?” in the system.

<table>
<thead>
<tr>
<th>What can I do?</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Starting up Smalltalk:</strong> a new user sees &quot;demos&quot; on the system menu, and reasons that a demo will be a good way to see what Smalltalk programs are all about. It turns out that the demos have some flashy graphics, but you just have to sit back and watch. The learner wonders where the demo programs are, but isn't sure how to find them. The programmer gives up on the demos, but remembers someone saying that classes are an important concept in Smalltalk, and that there is a &quot;browse classes&quot; choice on the main menu. When this is selected, the class hierarchy browser appears &amp;emdash. after making a few random selections here, the programmer sees what appears to be Smalltalk code, and hunkers down to read through it.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>How does it work?</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Analyzing cardhand display:</strong> a colleague has given the learner an example application, a blackjack game, to learn from. The learner plays with the game, and wonders how it does its display updates. The user is unable to find anything that looks like a &quot;display&quot; method in either the BlackJack class or its superclass CardGame. Eventually the learner discovers an &quot;open&quot; method that mentions a BJWindow class, and after looking there, finds a &quot;display&quot; method in its superclass ApplicationWindow. The learner can't make sense of what it does, and has no idea how to tell whether it is actually used by the blackjack application.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>How do I do this?</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Changing cardhand format:</strong> while playing with a blackjack game, a learner realizes that the cards are not being displayed effectively &amp;emdash. they are listed horizontally, and for a large hand, this means that not all are visible; a vertical listing would solve this problem. The learner remembers that the class hierarchy browser uses such a format, so decides to use it as an example. After reading a lot of code, the learner eventually discovers that the browser uses something called a ListPane, a class he or she remembers seeing once before, somewhere under Window. The learner locates this class in the browser and begins reading about it, trying to understand how it might fit into blackjack.</td>
</tr>
</tbody>
</table>

**Figure 4:** User interaction scenarios for learning Smalltalk. The italicized label names the high-level goal of the scenario, and the text describes the goals, understanding, and reactions of a user pursuing this goal in the Smalltalk environment.
What can I do?

5a. exploring demos helps new users learn by doing  
   (but scripted demos offer little for the user to do)  
   (but the demos may not be paradigmatic applications)  
   (but learners may have difficulty finding the corresponding code)

5b. exploring system tools helps new users learn by doing  
   (but the tools may be too complex for learners to understand)  
   (but learners may have difficulty finding the corresponding code)

5c. browsing and editing in the class hierarchy browser establishes core Smalltalk programming skills  
   (but may reduce time spent on instantiating and analyzing objects)

How does it work?

5d. the primacy of the class hierarchy directs attention to inheritance relationships among objects  
   (but learners may not be familiar with a class' superclasses)  
   (but it may reduce attention to to other important relationships, e.g. the model-user interface framework)

5e. the object communication summarized in the message execution stack allows learners to decompose application functionality  
   (but learners may not know how or when to explore the execution stack)  
   (but learners may be unable to interpret the communication patterns)

5f. the debugger's message execution stack supports learning transfer from procedural programming experience  
   (but may encourage learners to rely on a procedural model of computation)

5g. inspecting an object's instance variables across time supports mapping between its state and behavior  
   (but learners must first find or create a useful object to inspect)  
   (but learners may have difficulty managing and integrating many inspections of an object)

5h. analyzing an application with multiple system tools (e.g., as in the debugger) supports convergent reasoning  
   (but integrating information across tools may be difficult)  
   (but jointly managing the application and tool windows may be difficult)

How do I do this?

5i. interesting features of existing applications evoke subgoals  
   (but a particular goal may be difficult or impossible to pursue)

5j. existing applications and methods provide templates for the design of new functionality  
   (but learners may have difficulty finding or understanding examples)  
   (but users may be less likely to pursue other more appropriate possibilities)

5k. class and method names evoke analogical subgoal mappings  
   (but some spurious mappings may be pursued)

5l. navigating the class hierarchy supports unintentional learning  
   (but searching for specific classes can be frustrating)  
   (but the size of the hierarchy may intimidate learners)

Figure 5: Psychological claims embodied in the design of Smalltalk/V. The claims, all of which pertain to learning the Smalltalk language-environment, are structured into a principal upside, in roman type, and one or more associated downsides, in italic type, and are organized under the learner concern in the context of which they typically arise.
Each of these user concerns can be instantiated for the Smalltalk environment in one or more specific user interaction scenarios (see Figure 4). Such scenarios reify typical or critical usage episodes and can be generated both through empirical observation and from task analysis. Because scenarios provide a description of the artifact embedded in a context of user background knowledge, goals, and reactions, they are an appropriate framework for analysis of the artifact’s psychological consequences.

2.1 Extracting Learning Claims from Smalltalk

For illustration, consider the “Starting up Smalltalk” scenario in Figure 4. The system menu of Smalltalk/V offers “demos”, the selection of which allows the learner to run a half dozen animated graphic demonstration programs. Including any function in a system, and indeed offering it on the main menu, embodies claims about what is useful or desirable to users. Including “demos” on the Smalltalk system menu embodies the claim that the demos will be helpful in some way (illuminating, motivating, task-orienting, etc.) to learners pursuing the “What can I do?” concern (claim 5a in Figure 5).

Tradeoffs pervade all design discourse, and typically the claims we can infer from a design include associated “downsides.” Thus, the potential benefits of the demos are balanced by considerations that could limit these benefits: as the scenario conveys, the system may not support adequate exploration of the demos (and accordingly they could fail to be illuminating, motivating, etc.) or the demos themselves might be unrepresentative Smalltalk applications (and accordingly they could provide poor learning models).

Several aspects of this claim are important to note. First, it is not a claim about the intentions of the designers of Smalltalk, it is a claim about psychological consequences for users of Smalltalk. Our approach to design rationale does not attempt to externalize the designer’s reasoning process (cf. Conklin & Yakemovic, 1991), but rather attempts to externalize psychology embodied in the designed artifact and its use. We are pursuing analytical interests in the tradition of psychologists like Gibson (1979) and Simon (1981) who stress the causal role of the external environment in shaping behavior and experience. Second, the claim contains rich and specific information about a major design feature of the Smalltalk environment. Our objective is that the information be detailed enough to support further design work (in deliberate contrast to traditional guidelines which are often too general to directly guide design problem solving). Third, the claim is qualitative, informal and incomplete. For example, the psychological consequence “helps new users learn by doing” leaves it open as to whether the demos instruct the learner (that is, presents information), motivate the learner to seek information, or merely suggest a task-orientation toward learning Smalltalk (indeed there are other possibilities). In this case, we did not feel warranted in hypothesizing a more specific attribution of consequence.

Finally, the claim is an empirical hypothesis, in three distinct senses. First, it is part of our analysis of the psychological design rationale of the Smalltalk/V language and environment. As in any analysis, we could be wrong: the psychological science that warrants this claim could turn out to be wrong about the role of action and exploration in learning, or the science might be fine, but our appeal to it in this case could turn out to be erroneous, or — quite likely — both the science and our analysis are fine as far as they go, but overlook other, perhaps more important, psychological consequences of the demos (e.g., consequences that might only emerge from direct user studies of Smalltalk learning).

Second, to the extent that our analysis is correct, the demos claim is an empirical hypothesis made by Smalltalk as a theoretical entity. Elsewhere, we have argued that the designed artifacts of contemporary HCI are the most successful and appropriate theoretical entities in the developing science of HCI (Carroll, 1989a, 1989b, Carroll & Campbell, 1989). The demos claim codifies a part of the empirical theory of programmers and programming activity that is embodied in the design of the Smalltalk language and environment.

Third, the demos claim is part of our higher-level hypothesis that the level of design rationale depicted in Figure 5 is coherent. It is clear that an artifact as complex as Smalltalk
embodies a virtual infinity of psychological claims about its learners and learning situations. As in any other analytical project, our analysis must abstract and generalize in order to be codified at all. Our objective in constructing an analysis like that in Figure 5 is to capture a rich and coherent working representation of a design situation that is at the same time abstracted and abbreviated enough to be heuristic.

Note that the claims are not intended to be taken as privileged in any methodological sense. Psychological design rationale does not incorporate formal discovery procedures or any means of guaranteeing a priori that a given analysis is correct (no empirical science meets these positivist requirements, see Feyerabend, 1988). Indeed, it is likely that the possibilities for user experience and action codified in a psychological design rationale cannot ultimately be grounded in any “objective” epistemology (that is, any view of knowledge dissociated from meaningful situations). Our expectation is that the appropriate epistemology here (and elsewhere in psychology) must be relativized to the commitments people make towards contexts of experience and action (Polanyi, 1958; Schön, 1982; Suchman, 1987).

More broadly, we see neither the domain of our analysis (essentially cognitive psychology) nor its vocabulary as privileged. Many other analyses would be possible and may indeed be crucial (see below). The limitations of our analysis stem merely from our interests, the limits of our competence, and the incomplete state of our analytical project. We make this caveat to discourage positivist distractions for the reader and to lay our philosophical cards on the table. However, in this paper, our use of psychological design rationale is quite modestly empirical and focussed on its heuristic application in design argumentation.

This framework may be quite similar to others described in this special issue. Our notion of “user concern,” which we take to be an abstraction across more specific user scenarios, could also be seen as an abstraction of what Lewis, Reiman and Bell (1991) call “target problems”; their notion of “doctrine,” the knowledge one must have in order to use an artifact, seems to correspond to a subset of what we call claims (in purporting to be usable the artifact embodies claims that its users have or can generate its doctrine). One might also draw correspondence between our schema of “user concern, artifact feature, and psychological consequence” and MacLean, Young, Bellotti and Moran’s (1991) schema of “question, option, and criteria”. The key differences seem to be that our analysis is at a higher level (with respect to both artifact structure and psychology) and more bound to context of use: user concerns pertain to whole tasks, where questions pertain to constituent methods; artifact features and psychological consequences are analyzed as a set under the scope of a given user concern, not analyzed independently as are options and criteria. At some point, it will be important to pursue and resolve these connections, but we do not presume to be able to do that now.

To a great extent, psychological design rationale, as we understand it and have defined it above, merely systematizes extant practice. Many of the classic “case study” papers in HCI (e.g., Smith et al., 1982) are brimming with psychological claims, arguments and attributions (for instance, the key claim that graphical presentation of an interface metaphor can facilitate learning and using by supporting analogical reasoning). Perhaps even more impressive is the extent to which descriptions of computational artifacts are unwittingly psychological — many discussions of object-oriented systems, for example, include promises or claims about the “naturalness” of the paradigm for humans without any psychological account of what is involved in such an attribution (see Rosson & Alpert, 1990).

The approach we are pursuing seeks to improve this practice in two ways. First, case studies often innocently conflate the designer’s intentions with psychological consequences of the design as experienced by users. This is understandable in work that is both analysis and memoir, and surely designers’ intentions can provide key insights toward analyzing their work. But often there is a distinction: misconceptions and frustrations of users confound the best of intentions. Our goal is to help recognize and respect this distinction so
that each can play an appropriate role in understanding and guiding design. Second, the various claims, features, and attributed psychological consequences described in case studies often overlap and interact so intricately in the exposition that it can be difficult to see just what is being asserted and to extract general and applicable principles. Our hope is that claims analyses like that in Figure 5 can impose a discipline on the construction of psychological design rationales, to make it clearer what is being asserted and more feasible to apply what has been learned.

For example, claims 5c and 5d in Figure 5 capture a Smalltalk issue that has been discussed informally for years: the primacy of the class hierarchy browser (the system browser in Smalltalk-80; this tool provides access to and editing of Smalltalk code) both emphasizes the importance of browsing and editing code, and directs programmers’ attention to the inheritance hierarchy within which this code exists. Both of these are important consequences for users of Smalltalk: much of programming in Smalltalk consists of browsing and editing code, and understanding and taking advantage of inheritance relationships are key to the development of modular and elegant designs. However, making these tasks salient to novices tends to obscure other activities, for example, analyzing the functional relationships realized through an object’s instance variables (e.g., the relationship of the underlying application to its user interface).

The remaining claims in Figure 5 have a structure analogous to the two we have commented on. We will refer to them only as necessary in developing our design case study.3

2.2 Designing the View Matcher

Our psychological design rationale for Smalltalk with respect to the three learning concerns suggests many issues for redesign. Our consideration of these issues can be couched as a design hypothesis: a learning presentation of Smalltalk that preserves or enhances the constructive psychological consequences of the claims in the rationale, but that addresses the associated downsides of these claims.

For example, providing dynamic examples (like the demos) is a good idea, but it is undermined by lack of support for exploring their design and implementation and by the fact that they are not representative Smalltalk applications (users cannot interact with the demos; they are simply graphic scripts that “play out”, and are implemented as individual methods rather than as independent applications). Thus, we envisioned a presentation of Smalltalk that incorporated more paradigmatic examples and that better supported exploration of their design and implementation.

Offering a variety of interactive tools for analysis of running applications supports the “How does it work?” concern (see claims 5e-5h in Figure 5). Learners can open an inspector on an object to examine its internal state; in an error situation, they can analyze application functionality in the debugger’s message execution stack. They can open multiple tools to compare and contrast different information sources. But managing and integrating all these tools can be complex. This suggests a presentation of Smalltalk that permanently displays and synchronizes these tools with example applications.

The system debugger provides a model for this, coordinating an inspector and a class hierarchy browser with the message execution stack. However, particularly for a learner, the debugger can be overwhelming: it is typically encountered in the context of an error, it presents a lot of information about message activity, object states, and implementation, and it is only loosely coordinated with the user’s application. This suggests a design that more tightly couples the application with the system tools: incorporating a permanent application

---

3 Our development of psychological design rationale, our understanding of the psychological design rationale for Smalltalk, and our design of the View Matcher for learning have developed in concert since 1988; the presentation here reflects the current state of all this work. A brief description of our Smalltalk analysis circa 1988 can be found in Carroll and Bellamy (1989); a description of the View Matcher circa 1988 can be found in Carroll (1990b, pp. 267-272).
**What can I do?**

*Starting up View Matcher:* a new user sees "blackjack" on the View Matcher menu, and reasons that this might be a fun and useful way to learn about Smalltalk programs. When the game starts, some classes that seem to be involved in the game (e.g. BlackJack, Card) appear. By selecting these classes, the programmer is able to see what appear to be message names and code, but is unable to make much progress with it. After playing the game for a while, the user wonders what the empty panes are for, goes back to the menu and experiments with "halt mode". Now when the game is played, it stops in the midst of doing things, and what seems to be a list of messages (the same ones that he or she saw earlier in the browser) is shown. Selecting a message in the list, the learner notes that the other parts of the display are updated, and settles down to try to make sense of all the information.

**How does it work?**

*Analyzing cardhand display:* the new user wonders how blackjack displays its cards, and has noticed that in halt mode the game stops before each display update. The user plays the game to such a point, then looks at the information displayed in the various panes. By selecting the top message on the stack (:hp2.showHand:ehp2.) and reading its associated commentary and code, the programmer sees that this message is sent to the card hand object when it is time to display its cards. The user infers that this message was sent in the course of processing the next message in the stack, just as would be true for a call stack. The rest of the messages on the stack seem very confusing, involving a number of non-blackjack classes (e.g., TextPane, NotificationManager), but using the commentary the programmer is able to see how the responsibility for displaying things is distributed between the window and its components, and the blackjack and its components.

**How do I do this?**

*Changing cardhand format:* while playing the blackjack game, the programmer realizes that the cards are not being displayed effectively &emdash. they are listed horizontally, and for a large hand, this means that not all are visible; a vertical listing would solve this problem. In halt mode, the user discovers that just prior to displaying the player's hand, the game stops, and that the top message on the execution stack is `showHand`. Selecting it causes the browser to locate the associated method code. The user tries making a change to the code, confirms by resuming the game that the format has in fact been changed, and then iterates through this process to get the format to look right.

**Using the full system.**

*Building an address book:* after working with blackjack in the View Matcher, the learner decides to try a new application, an address book, using blackjack as a model. Reasoning by analogy, the programmer figures that the address list will be like the card hands, but that there will be just one of them. The programmer goes to the classes used by the blackjack game, borrowing and editing code that seems relevant. When the user first tries to start up the new application, an error message box appears. The user is relieved to see the familiar message execution stack. After looking at the stack, though, the user is distressed to find many messages on the stack that were never encountered in the View Matcher explorations, and no commentary to help make sense of it.

**Figure 6:** Learning scenarios for the View Matcher. The first three scenarios instantiate the same high-level goals as those presented in Figure 4, but describe pursuit of the goal in the View Matcher environment. The fourth illustrates transfer of learning from the View Matcher to standard Smalltalk.
view, tailoring the information presented in the tools to be application-relevant (e.g., a class hierarchy that displays only classes and methods used by the application), and providing help texts to support integration across tools.

The design of the View Matcher was elaborated by developing specific scenarios of use. The example scenarios in Figure 6 depict learners beginning with high-levels goals similar to those of the earlier scenarios, but the user’s actions and experiences in pursuing these goals are those afforded by the new system. In “Starting up View Matcher”, the coordination among views has been further specified: when the learner opens the blackjack game, the class hierarchy updates to show a list of classes, promoting the inference that these are the objects involved in the application. The start-up scenario also illustrates how the design enhancements combine to support the learner — the availability of a familiar application in combination with the other panes in the tool suggest activities to the learner.4

The fourth scenario in Figure 6 represents a new kind of concern that inheres in any learning environment: at some point, the learner moves on to work (and continue learning) in the environment. When this happens, the learner may re-experience the first three concerns (“What can I do?”, “How does that work?”, “How do I do this?”), but now in the context of the target environment, and with the View Matcher learning experiences as background knowledge. In this sense, “Using the full system” can be seen as a sort of meta-concern — how do users do the things they were able to do before, and how do they go on from there? An important part of our design hypothesis was that by providing paradigmatic example applications, and by building the coordinated views out of standard system tools, we would ease learners’ transition from the View Matcher environment to normal Smalltalk programming.

<table>
<thead>
<tr>
<th>What can I do?</th>
</tr>
</thead>
<tbody>
<tr>
<td>7a. exploring paradigmatic interactive applications helps new users learn by doing</td>
</tr>
<tr>
<td>(but users may spend too much time using the application and not enough time learning about it)</td>
</tr>
<tr>
<td>7b. permanent display and coordination of an application's multiple views facilitates learning by doing</td>
</tr>
<tr>
<td>(but the multiple views may provide too complex a representation of the application)</td>
</tr>
<tr>
<td>7c. a filtered class hierarchy encourages opportunistic analysis of the objects in an application</td>
</tr>
<tr>
<td>(but users may be frustrated if they want to examine hidden classes)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>How does it work?</th>
</tr>
</thead>
<tbody>
<tr>
<td>7d. permanent display and coordination of multiple views supports convergent reasoning</td>
</tr>
<tr>
<td>(but does not develop skills for accessing and managing system tools)</td>
</tr>
<tr>
<td>7e. the application episodes defined by view matcher breakpoints are coherent units of analysis</td>
</tr>
<tr>
<td>(but may interrupt normal interaction)</td>
</tr>
<tr>
<td>(but it may be difficult to make sense of an application in the midst of a message send)</td>
</tr>
<tr>
<td>7f. the object communication summarized in the message execution stack allows learners to decompose application functionality</td>
</tr>
<tr>
<td>(but learners may be unable to interpret the communication patterns)</td>
</tr>
<tr>
<td>7g. the debugger's message execution stack supports learning transfer from procedural programming experience</td>
</tr>
<tr>
<td>(but may encourage users to rely on a procedural model of computation)</td>
</tr>
</tbody>
</table>

4 Additional scenarios are discussed in Carroll (1990b, pp. 270-272) and in Carroll, Singer, Bellamy, and Alpert (1990).
7h. application-specific commentary for each message in the stack evokes an application
-oriented interpretation of object communication
  (but this understanding may not generalize to other applications)
7i. synchronizing the browser with the method selected in the stack directs attention to the
  code relevant to the application's current state
  (but it may discourage exploration of methods not on the stack)
7j. joint updating of the matched views temporally groups information, helping users parse
display
  (but too much information changing at once might be confusing)
7k. an animated message stack serves as an advance organizer for the halted stack
  (but learners may be frustrated by not being able to explore the animated stack)

How do I do this?
7l. concrete, familiar applications will evoke specific subgoals
  (but a particular subgoal may be difficult or impossible to pursue)
7m. examples that misbehave provide intrinsic motivation for analysis
  (but may encourage too narrow a focus in analyzing the example)
7n. coordination of an application's multiple views simplifies access to task-relevant
  information
  (but learners may come to rely on this coordination)
7o. modifying an application is easier than creating one from scratch
  (but the existing application may not be consistent with the user's current view of
  object-oriented design)

Using the full system
7p. finding and modifying code encourages code reuse as a programming paradigm
  (but it may undermine other reuse strategies like subclassing)
7q. experience with system tools in the view matcher transfers to their use in non-view
  matcher activities
  (but user may be frustrated by the absence of coordinated views)
7r. paradigmatic examples help users construct principles of good design
  (but learners may overgeneralize the details of particular examples)
7s. understanding/debugging examples generalizes well to other programming tasks
  (but may induce too narrow a view of object-oriented programming)

Figure 7: Psychological claims embodied in the View Matcher for learning. The analysis
is organized by the three general learning concerns used in the original analysis of
Smalltalk, plus a fourth concern typifying a learning transfer situation.

2.3 Psychological Design Rationale for the View Matcher
The View Matcher system is the end-state of the design hypothesis. It was not
generated by the design rationale in Figure 5, but then “deductive invention,” more than
merely an oxymoron, is probably a technological contradiction (and indeed, here we really
do depart from Bacon’s optimism about practical science; see Carroll, 1989a). The design
of the View Matcher was heuristically guided by the claims analysis of Smalltalk. As the
scenarios representing the new design are elaborated and implemented, they then serve as
an organizing rubric for analysis of the incipient system (see Figure 7).

The claims embodied in a new artifact can often be seen as evolving from claims
analyzed for the precursor artifacts. A case in point is claim 7a concerning the role of
paradigmatic interactive applications in learning by doing. Figure 8 illustrates how this
claim evolved from our analysis of the demos in Smalltalk/V (claim 5a in Figure 5).
Instead of the demos, the View Matcher offers familiar applications that are interactive and
that were crafted to exemplify object-oriented decomposition and the separation of application model and user interface. By embedding a paradigmatic example in a tool designed to guide learners’ analysis, the View Matcher both exploits the positive aspects of the demos claim and addresses its negative aspects. Of course, it is still possible that our design of the examples was more quirky than we thought, or that the View Matcher supports exploration inadequately. However, the design rationale oriented us to these issues and at least we made some improvement.

It is typical for design enhancements to introduce new downsides. Thus we see in claim 7a that at least one risk of incorporating example applications that are familiar and interactive is that they might be too engaging — learners may get so caught up in their use of the application that their motivation to understand how it works is diminished.

usage situation:
procedural programmer starts up Smalltalk first time, tries out graphics demos

consequences of demos:
aids learning by doing, but is not interactive or paradigmatic, and code not easily accessible

capitalize on demos consequences:
offer paradigmatic interactive examples, with coordinated views to support analysis

new usage situation:
procedural programmer starts up View Matcher first time, opens the blackjack game

consequences of examples:
improved support for learning by doing, but example could be too engaging

Figure 8: Evolution of demos claim. The figure schematizes an example of our general reasoning heuristic, to capitalize on the constructive aspects of claims while addressing their downsides.

A similar evolution can be seen in the View Matcher’s emphasis on the message execution stack. The Smalltalk analyses pointed to the psychological benefits of the stack representation for users pursuing the “How does it work?” concern (for decomposition of application functionality, and for learning transfer, see claims 5e and 5f in Figure 5). In the View Matcher, the stack is the controlling view for application analysis: when the application is halted at a breakpoint, the current message execution stack is displayed, and selection of a message in this stack causes the class hierarchy browser to display the corresponding method code, the inspector panes to display objects used by that method, and the commentary pane to present an explanation of the method’s role in the application transaction being processed (see Figure 3). Increasing the salience of the stack in this way strengthens claims made by Smalltalk (claims 7f and 7g). At the same time, problems of stack interpretation are mitigated by providing the explanatory text (claim 7h). Of course, encouraging learners to develop an application-specific interpretation of message patterns in
the stack increases the risk that the understanding they develop will be too specific to
genralize to other situations.

Many of the claims in Figure 7 reflect the consequences of integrating and coordinating
multiple views onto an example application. This design feature eases the downsides of
many of the original Smalltalk claims — the relative lack of support for exploring the
demos, the difficulties in managing the system tools, the overattention to inheritance
relationships (see claims 5a, 5d, 5e, 5g, and 5h in Figure 5). However, the coordination
also introduces new claims with new downsides. Thus the fact that the tools all provide
information about the example application, and that they are jointly updated when the user
explores the application encourages users to try to make sense of the information in terms
of their personal use of the application (claim 7b); the downside is that a great deal of
information inheres in the multiple views, and this may be overwhelming to a learner.

As another example, consider the use of breakpoints to coordinate the application with
the message execution stack: when the user interacts with the application, it halts at
predefined points (user input events and display updates). This feature addresses the
downsides of claims 5e and 5g in Figure 5; it removes the need for the learner to initiate use
of the debugger and inspector information. But it also embodies a new claim, that the
application episodes defined by the breakpoints are coherent units of analysis (claim 7e in
Figure 7). The downside is that natural interaction with the application is interrupted, and
this may add to the difficulty of making sense of the application.

In some cases, claims from the original analysis of Smalltalk were dropped from the
new analysis. Incorporating a class hierarchy browser as just one of the several view-
mated representations reduces the salience of this view (and hence the emphasis on
browsing and on inheritance relationships, see claims 5c and 5d in Figure 5). Part of
coordinating the class hierarchy with the example involves filtering it to display only
application-relevant classes and methods. Users can still navigate in the browser
themselves, but because the number of classes is relatively small, the amount of
unintentional learning possible is considerably reduced. Indeed we felt that it was reduced
enough to leave this claim (see claim 5l in Figure 5) out of the View Matcher analysis; our
goal was to include only the leading claims of the new artifact. Again, however, the
changes to the class hierarchy view introduce a new claim. By limiting the browser to
information about application-relevant objects, users may be more likely to engage in
opportunistic analysis of these objects (claim 7c in Figure 7). The new downside is that
they may come upon the names of classes or methods that are not in the filtered view (e.g.,
generic or inherited functionality), and be frustrated at their inability to explore them.

These examples illustrate how working with a set of claims propagates effects: claims
disappear or change in prominence, new claims appear, and new tradeoffs are provoked.
As our design work proceeded, claims were continually being recognized, and new
tradeoffs and design issues caused us to realize new possibilities. For instance, the joint
updating of the View Matcher tools temporally groups related events in the interface and
helps the learner spatially parse the information display (claim 7j in Figure 7). Given the
number of simultaneous views in the View Matcher, this is an important psychological
consequence, but it was introduced as a side-effect of coordinating the multiple views.
Another case was the development of an animated view of the message execution stack
(claim 7k). Our formative evaluations of the View Matcher suggested that some learners
interpreted the message stack as a history list of messages sent (and not as a list of methods
currently suspended at the point of a halt). We introduced an animated view of the
execution stack in which the application’s message-passing is never halted (and therefore
cannot be analyzed in the midst of its execution). Watching the stack grow and shrink in
response to user requests prepares learners for their subsequent exploration of the (halted)
stack; the downside is that learners may see something of interest happening in the
animated stack and feel frustration at being unable to pursue it.

The claims listed in Figure 7 under “Using the full system” differ in an interesting way
from those we have discussed to this point. The relevant scenarios in this case involve the
use of the target system, not the View Matcher (see “Building an address book” in Figure 6). Thus, the claims reflect design characteristics of one artifact (the View Matcher) that have consequences for use of another (standard Smalltalk/V). We remarked earlier that such scenarios inhere in any instructional system, but in fact, any scenario reflecting learning transfer from one artifact to another would have the same character.

The View Matcher provides a good example of Carroll and Kellogg’s (1989) point that designed artifacts embody a “nexus of psychological claims,” not a list of independent psychological atoms. As in any other view of design, it is not possible to bound the scope of effect of design interventions. The effects of particular design characteristics (e.g., the use of paradigmatic examples, the coordination of the system tools) have wide-reaching psychological consequences. However, the empirical “validity” of a specific claim can be examined: the permanent display and joint updating of the View Matcher tools either does or does not support the integration of the information displayed in tools (say, relative to the baseline integration supported by the Smalltalk environment); specific error patterns can indicate whether learners become overwhelmed with the amount of information the View Matcher presents, or whether experience with the tools as integrated by the View Matcher transfers at all to using the tools as they are presented in the Smalltalk environment.

Because of the interconnectedness of the various claims, such evaluations will always be somewhat inconclusive (though their heuristic value can be leveraged if a diversity of evaluations are jointly interpreted against the rubric of a psychological design rationale). Indeed, this consequence for evaluation clarifies pervasive problems with summative evaluation. Aggregate summative evaluations have served little use in HCI design because they merely order the measured “usability” of whole artifacts without involving descriptions of the artifacts that could support credit/blame analysis (Carroll, 1989b). The further caveat from design rationale is that this problem may not be mitigated by making analytic summative evaluations (e.g., feature by feature assessments of an artifact) because the various features participate in interconnected claims (e.g., evaluating a command name — or even a command language paradigm — may be uninformative unless one also evaluates the interface display metaphor and the semantics of the user’s task; Carroll, 1985).

3. A VIEW MATCHER FOR CODE REUSE

When we began to deploy the View Matcher as part of our minimalist curriculum for Smalltalk, several of our users inquired as to whether we could extend the tool for use in routine programming. These users did not make very directive requests, rather they seemed to appreciate something about the View Matcher “approach” (interactive paradigmatic examples and coordinated views of jointly displayed tools, including interactive execution tracing). We ultimately decided that this was indeed an interesting request to try to respond to, perhaps in part because it required us to discover the details of the request itself.

We decided to focus our redesign effort on facilitating code reuse, because it is so fundamental to Smalltalk programming; Smalltalk offers a rich and extensible hierarchy of reusable classes, and a large portion of any programming project is the discovery and application of existing code. The general reuse task we are addressing is one in which the programmer has a goal in mind and wants to reuse code insofar as this is appropriate. The programmer engaged in this task may have several sorts of concerns (see e.g., Fischer, 1987; Raj & Levy, 1989). First, he or she may work at identifying a candidate class or classes in the existing hierarchy, pursuing the information retrieval concern of “What can do this?”. At some point, a candidate class will have been identified, and at this point the concerns become oriented specifically to that class: the programmer wonders “What can it do?”, to see if it does have the desired functionality. If the class does indeed look promising, then the programmer wonders “How is it used?”, to reason about how objects of this sort could be used in the project at hand. If problems develop in trying to use the class, the programmer will wonder “How does it work?”, so that the problems can be
debugged and resolved. Finally, if the programmer determines that the class has some, but not all of the necessary characteristics, the reuse project will encompass the “How to extend it?” concern. Our initial design work has focussed on the middle three concerns, those involving questions about a target class’ functionality, usage protocol, and implementation.

<table>
<thead>
<tr>
<th>What can it do?</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>What Sliders do:</em> A programmer sees Slider in the class hierarchy, and the name sounds like it might be useful to the current project, a color-mixing application. It is a subclass of ControlBox, which seems consistent with this inference. The programmer takes a look at the messages defined for Slider; they sound as if they involve input handling (e.g., <code>adjustToMouse:</code>), but it isn’t clear how they fit together, or what the look and feel of the Slider will be. The programmer tries to find an example instance to work with, asking for “allInstances” of the class, but none exist. Because this is a user interface object, the programmer suspects that it will require considerable set-up to create an example, so finally hunkers down to read through the code.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>How is it used?</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Hooking up a GameBoardPane:</em> A programmer wants to use an instance of GameBoardPane in a chess game being developed, and needs to know how to connect it to the current objects. The programmer takes a look at the code for GameBoardPane, and sees that it has a <code>squares</code> instance variable, and asks for “senders” of the <code>squares:</code> message, hoping to see situations in which this variable gets set. A long list is returned, but the programmer infers that many of them are for other implementations of this message. Scrolling through the list, the programmer sees GameBoardWindow. This class sounds promising, so the programmer goes off to look at its code, to try to figure out how it uses a GameBoardPane.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>How does it work?</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Analyzing slider scale conversion:</em> The programmer has successfully incorporated three sliders into the color-mixing application, setting them up to have a horizontal lay-out. As the mouse moves, the slider changes size, but the color output isn’t right. The programmer puts a halt into the code for <code>adjustToMouse:</code>; and when the application halts, inspects the mouse value, which seems correct. Using the debugger stepping functions, the programmer then traces through all subsequent message sends, checking values at each point to see when the error occurs.</td>
</tr>
</tbody>
</table>

Figure 9: Reuse scenarios for Smalltalk/V. The scenarios exemplify user concerns that can arise after a programmer has identified a candidate class for reuse.

Like the learning concerns, the reuse concerns can be instantiated as specific user scenarios. The scenarios in Figure 9 instantiate these general concerns for the Smalltalk environment, and suggest some of the psychological consequences that Smalltalk might have for an experienced programmer interested in reusing a particular class. An analysis of these consequences, combined with our analysis of the previous View Matcher work, formed the starting point for our design of a View Matcher for code reuse.

3.1 Extracting Code Reuse Claims from Smalltalk

The key differences between the learning and the reuse situations are that the goal of the reuse situation is well-defined, and the user is assumed to be an experienced programmer fluent at finding and interpreting information in the system. However, analyzing a class in order to reuse it can be seen as a specialized case of learning (the programmer seeks to learn about a particular class or functionality), and the Smalltalk
claims we constructed for the reuse situation (see Figure 10) reflect similarities between the two task situations. For example, if an instance of a target class is available, a programmer can send messages to it to explore its functionality, just as a learner might explore a demo application to learn something about how Smalltalk programs work (claim 10a). As the first scenario in Figure 9 illustrates, the downside is that the programmer must often create objects from scratch, and the initialization of complex objects (e.g., a user interface object that must be connected to a number of other objects) can be very cumbersome (cf. Gold & Rosson, 1990).

10a. sending messages to an instantiated object supports discovery of its functionality
   (but finding or creating a representative instance may be difficult)
   (but trying out individual messages may be tedious and distracting to ongoing work)
10b. class and method names suggest the functionality provided by a class
   (but some names are ambiguous or inappropriate)
   (but the behavior may be too complex to suggest with a name)
10c. the primacy of the class hierarchy directs attention to inheritance relationships among objects
   (but programmers may not be familiar with a class' superclasses)
   (but superclass functionality may not predict subclass specialization)
10d. tracing an application's message passing allows programmers to build schemas for use of its components
   (but temporal integration of requests to a single object may be difficult)
   (but abstraction of communication and control relationships among the objects may be difficult)
10e. halting an application in the midst of receiving a particular message supports a functional analysis of the message's role
   (but it may be difficult to tie this analysis to the visual state of the application)
10f. the primacy of the class hierarchy directs attention to inheritance relationships among objects
   (but it may reduce attention to sender/receiver relationships)
10g. browsing a class' method code helps in understanding how to use the class
   (but code specifies an implementation, not a usage situation)
10h. message-naming conventions allow programmers to predict parts of a class' protocol
   (but many messages are not covered by convention)
10i. examining the "senders" of a message supports analysis of the context in which this request is made
   (but a senders listing may include many false alarms)
10j. tracking the messages sent, and the state changes produced in the process of evaluating a message allows users to analyze message implementation
   (but the programmer must initiate, control, and integrate the analysis)
10k. the primacy of the class hierarchy directs attention to inheritance relationships among objects
   (but it may reduce attention to other functional relationships, e.g. those provided through an object's instance variables)

Figure 10: Reuse claims embodied in the design of Smalltalk/V.

As in the learning scenarios, if programmers can find or create a representative example, Smalltalk encourages them to enlist the system tools in the analysis of the example, in this case to address concerns of “How is it used?” or “How does it work?”. Using the debugger, programmers can trace the message-passing activity of an application that uses a target object, to see when and in what context messages are sent to the object (claim 10d), as well as to see what messages the target object sends in fulfilling these
requests (claim 10j); the inspector allows them to examine the target object’s changing state over time, to understand how the object manages itself in the process of fulfilling requests (claim 10j). But also as in the learning scenarios, it is up to the programmer to set up an informative situation, and to extract and integrate the information relevant to the use or implementation of the target object.

Again as recorded in our learning analyses, the primacy of the class hierarchy focuses programmers’ attention on inheritance relationships. This has consequences for each of the three reuse concerns (claims 10c, 10f and 10k). For programmers wondering whether to use a class, an inheritance context can be quite helpful: if they are familiar with the superclass’ functionality, they should be able to predict something about the target class’ functionality. In contrast, if a programmer wishes to know how an object is used, inheritance relations will help little; what the programmer really needs to know is how other kinds of objects make requests of this object.

The “Hooking up a BoardGamePane” scenario in Figure 9 illustrates a programmer pursuing this question, asking for the “senders” of the squares: message. By looking at the code within which a message is sent, the programmer can determine something about the sending context (e.g., what kinds of objects send it, and when; claim 10i). The downside is that in Smalltalk, a given message name may have multiple implementations (i.e., in different classes), and the implementation invoked in response to a message can only be known at runtime. The senders query returns a list of all code that includes the message string, and this list will often contain many false alarms — code in which this message is sent, but not to an instance of the target class.

3.2 Redesigning the View Matcher

Our psychological design rationale for Smalltalk with respect to the code reuse concerns, in concert with that for the View Matcher for learning, focused our consideration of issues for redesign. Again, we worked with these issues by envisioning a design hypothesis. Given our starting goal of applying the View Matcher concept to the code reuse domain, much of our design reasoning was by analogy: what are the analogous representative examples to incorporate into a reuse analysis, what should the multiple views of that analysis convey, and how should they be coordinated? To some extent, our ability to reason by analogy in this way indicates that the issues we are concerned with are general — general user propensities and requirements, general limitations of the Smalltalk language-environment, and general opportunities for redesign. But given our a priori commitment to the View Matcher framework, it also probably reflects the bias in our analysis (of course, this is what genre and style in design are all about! See Newman, 1988).

The design rationale for learning Smalltalk and for the original View Matcher highlighted the importance of example applications as vehicles for learning, and pointed to issues concerning support for analyzing such examples. Similarly, the design rationale for reuse in Smalltalk highlighted the potential of analyzing example applications’ usage of a target object, understanding the context of this use, and analyzing the target object’s state and activity over episodes of use. By analogy, then, we hypothesized that our new View Matcher should revolve around an example application that makes use of a target object, and have constituent views that support an analysis of this use.

Figure 11 illustrates how our design of the reuse examples considered not only the advantages of working with an example instance of the target class (claim 10a), but also the claims concerning demos and interactive examples recorded in our prior learning analyses. Users first learning Smalltalk are learning by doing — searching for reasonable goals, pursuing goals opportunistically, trying to make sense of things. Smalltalk’s demos encourage this mode of learning, but are not paradigmatic and do not offer much for learners to do (claim 5a in Figure 5); the View Matcher addresses this downside with its provision of paradigmatic interactive applications (claim 7a in Figure 7). But the code reuse situation is different. The experienced programmer seeking to reuse a class is not
looking for goals; rather the programmer is in the midst of a meaningful project and wants a question about reuse answered in a way that distracts as little as possible from the ongoing project. This suggests that the examples for reuse analysis should be scripted (as the Smalltalk demos are) — usage vignettes that exemplify paradigmatic uses of an object, but that require no work other than interpretation.

**Figure 11:** Evolution of demos and examples in the View Matcher. The figure schematizes how the usage vignettes in the reuse View Matcher evolved both from the analysis of reuse in Smalltalk and from the earlier analyses of Smalltalk learning.

Another key feature of the reuse View Matcher is what we term a “usage view”: the usage vignettes are decomposed into typical usage episodes; these in turn are decomposed into messages involving the target class (see Figure 12). This view corresponds to the stack view of the learning View Matcher, in that users’ interactions with it drive the coordination of the other views. In the View Matcher for learning, the message execution stack supports learners’ decomposition of running application into its basic transactions (via breakpoints delineating interactions between the underlying application and its user interface), helping them to see how the parts cooperate to produce the overall functionality (claim 7e and 7f in Figure 7). The stack also provides an important path for learning transfer, though perhaps promoting too procedural a view of application activity (claim 7g). In the reuse case, we assume experienced programmers, so the transfer consequence becomes less important. And while the debugger does allow programmers’ to analyze the messages sent to a target object and their effects (claims 10d, 10c and 10j in Figure 10), it is left to the programmer to create, manage and integrate illuminating tracing episodes. We hypothesized that by providing an analysis of the vignettes into usage episodes typifying the target object’s contribution to the application, the tool would support a decomposition that was both more task-relevant and more object-oriented.
Figure 12: A View Matcher window for the BoardGamePane class. The usage view is in the upper right, the communication map is in the upper left, the class hierarchy view the lower right and the commentary view the lower left. The user has already viewed the Gomoku usage vignette, and is exploring the messages involving the BoardGamePane in the “starting up the game” episode. In the communication map, the connection between the window and the subpane has been selected, and shows the name of the window’s instance variable that refers to the instance of BoardGamePane.

As in the case of the View Matcher for learning, many details of our design hypothesis capitalize on the claims of Smalltalk for reuse, while addressing their downsides. Thus the usage view is an approximation to the message tracing available via the debugger; it enumerates only messages that are sent to the object of interest or that include it as an argument. We added to this message trace a presentation that would make explicit the communication and control relationships among the participating objects (the “communication map” in Figure 12).[^5]

Finally, we hypothesized that coordinating the inspector with specific requests made to the target object would provide a meaningful structure in which to understand an object’s changing state over time.

By including a filtered class hierarchy browser as another coordinated view, we are able to maintain the advantages that the inheritance structure brings to predicting a target class’ functionality, while addressing the downside that inheritance has little to say about usage. Because the example application exemplifies use of the target object, exploration of it in the browser will necessarily involve accessing true “senders” code.

[^5]: The antecedents of these representations were developed by Rosson as discussion aids in her object-oriented design workshop at the Watson Center.
What can I do?

*What Sliders do:* a programmer sees Slider in the class hierarchy, and the name sounds like it might be useful to the current project, a color-mixing application. The programmer opens a View Matcher on it, and selects the first example, a football player analysis program. A short demo of the football program is shown, and the programmer sees that sliders are being used to manipulate player characteristics that predict several player success measures. The programmer recognizes that this situation is very similar to the needs of the color-mixer, but goes on to view a second vignette, an economic simulation in which the sliders interact with one another, constraining each other’s values.

How is it used?

*Hooking up a GameBoardPane:* a programmer wants to use an instance of GameBoardPane in the chess game being built, and needs to know how to connect it to the game objects. The programmer opens a View Matcher on the GameBoardPane class, and selects the first example, a Gomoku game. After watching the demo, the programmer examines the object communication map; the map shows that there is a Gomoku object which points to a GameBoardWindow, and that the window then points to the GameBoardPane. The programmer wonders where the board comes in, and selects the “draw the board” usage episode; the object communication map updates to show that in this example, the information about the board comes through the Gomoku and window connections.

How does it work?

*Analyzing slider scale conversion:* a programmer has successfully incorporated three sliders into the color-mixing application, setting them up to have a horizontal lay-out. As the mouse moves, the slider changes size, but the color output isn’t right. The programmer opens a View Matcher on Slider, selects the football example, and expands the “increase player speed” episode. The usage analysis updates to show the messages sent to the slider. The programmer expands the *adjustToMouse:* message and sees that it involves an internal request for a scale reading. The programmer examines the code for the scale reading, finding an error in the conversion algorithm.

**Figure 13:** User scenarios for the reuse View Matcher. The scenarios instantiate the same high-level goals as those used for analysis of Smalltalk in Figure 9, but describe how the goal might be pursued using the new system.

The design of the reuse View Matcher was elaborated by developing specific scenarios that instantiate the code reuse concerns. As the scenarios in Figure 13 illustrate, a programmer first opening a View Matcher on a target class can view one or more usage vignettes (e.g., a football analyst that uses sliders). Each of these vignettes can then be expanded into its associated usage episodes (e.g., “starting up the game” in the Gomoku vignette, see Figure 12), and these episodes can be expanded to display the requests that are either sent to the target object directly (e.g., *aPane drawBoard:* *(owner board)*), or that include the target object as an argument. Finally, each message sent to the target object can be expanded to display messages the object sends to itself in carrying out the request (e.g., *self drawGrid:* *aGameBoard*).

The other views of the example application are coordinated with programmers’ interaction with the usage view. The communication map provides a graphic representation of the relevant communication context: At the level of a usage vignette, this consists of the relationship of the target object to the controlling objects of the application (e.g., illustrating that *boardPane* is an instance variable of the BoardGameWindow object, which in turn is an instance variable of the Gomoku object). When a usage episode or a specific message within an episode is selected, the communication map updates as necessary to include other
supporting objects. At any time, the user can select one of the objects in the communication map, and inspect that object’s internal state either before or after the point under analysis in the usage view (e.g., before an episode begins or after it completes, before a particular message is sent or after it is evaluated).

14a. viewing paradigmatic scripted demos that use an object helps users analyze its functionality
   (but the concept induced might be too narrow)
   (but users may have difficulty isolating the target functionality)
14b. class and method names suggest the functionality provided by a class
   (but some names will be ambiguous or inappropriate)
   (but the behavior may be too complex to suggest with a name)
14c. permanent display and coordination of multiple views of an object's usage supports convergent reasoning.
   (but the novelty of some of the views and the amount of information presented may be difficult)
14d. the connections between the target object and other objects in a usage episode provide templates for its incorporation into other designs
   (but may convey a fragmented view the demo application's design)
14d. the requests made of an object during a usage episode provide templates for exercising its functionality.
   but may result in a fragmented view of the application using it)
   (but programmers may overgeneralize the details of particular episodes)
14f. the pattern of requests made to an object during a usage vignette evokes specific subgoals that drive analysis of its role in the demo
   (but it may be difficult to connect analysis subgoals to features of the demo)
14g. directing programmers attention to the code in which a message is sent facilitates creation of analogous sender code in a new application
   (but experienced users may be confused when the browser doesn't display message implementation)
14h. result-oriented commentary for each message in the usage view promotes a functional analysis of the target object's role
   (but may discourage analysis of how a particular request is fulfilled)
14i. seeing the usage vignette first serves as an advance organizer for the usage and communication views
   (but programmers may forget some features of the demo)
14j. analyzing requests made to self in response to a message supports inferences about message implementation
   (but it may discourage a complete decomposition the request)
14k. examining the state of relevant objects before and after a message send facilitates analysis of the message's consequences
   (but some state changes may take place during message evaluation)

Figure 14: Psychological claims embodied in the View Matcher for code reuse.

The filtered class hierarchy view is used to show the relevant sender code: when an episode is selected, the browser shows the method code from which this episode was initiated (e.g., the Gomoku newGame method for the “starting up the game” episode); when a particular message in the episode is selected, the browser displays the code within which this message was sent (e.g., drawPieces: is sent within the BoardGameWindow redrawBoard: method). The commentary describes the contributions of the target object as a function of the current selection in the usage view, either to the application as a
whole, more specifically to a usage episode, or even more specifically to the evaluation of a particular message.

3.3 Psychological Design Rationale for the Reuse View Matcher

We have implemented the View Matcher for code reuse (Rosson, Carroll & Sweeney, 1991a;b). We are continuing to explore the use of psychological design rationale by constructing a claims analysis for the system, to guide our ongoing prototyping and evaluation work. This interim analysis is presented in Figure 14.

Some of the claims involve familiar features and consequences. For instance, the canned demos are paradigmatic examples of reuse of the target object (claim 14a). But they suffer from the downside of any set of examples; they can be incomplete in the functionality or usage protocols that they exemplify. Further, the decision to exemplify target objects from the reuse perspective leads to another downside: users viewing the vignettes may have difficulty identifying just what functionality is contributed by the target object. Notice though, that one of the original downsides analyzed for the Smalltalk demos in the learning situation, that scripted demos offer little for the learner to do (see claim 5a in Figure 5), does not appear in this analysis. As we argued earlier (see Figure 11), experienced programmers seeking to reuse code are not looking for extra things to do, so the scriptedness of the reuse examples is less likely to entrain this negative consequence.

Other claims reflect the consequences of new features — the usage episodes, the communication map, and the sender-oriented class hierarchy view. The usage episodes and the communication map document usage situations which can then serve as templates for reuse in new situations (claims 14d and 14e). But these representations are filtered views of the design and message activity of the vignettes, and there may complex or eccentric situations in which mapping from the template to the new problem is indeterminate. The usage view also suggests a subgoal decomposition of how the target object is used (claim 14f, which is analogous to claim 7f concerning the message execution stack). The class hierarchy view supports a copy/edit reuse strategy (Lange and Moher, 1989) by directing attention to the code in which a message is sent (claim 14g), but may confuse programmers who expect to see the implementation of the message itself. These risks are salient to us at this stage of design, because we are experimenting with representation and analysis tools that depart significantly from the standard environment.

The evolution of the reuse View Matcher affords a better understanding of application genre. Our analyses capture the similarities between the two View Matcher designs: both capitalize on the psychological consequences of learning from paradigmatic examples, and of reasoning from multiple representations. One can see these similarities as defining a View Matcher genre. Each design also refines the genre as a function of the situation in which the learning occurs: for novices, the examples are interactive and users are encouraged to decompose them into the major transactions between the underlying application and the user interface. For the reuse situation, the examples are non-interactive and users are encouraged to develop object-specific usage analyses.

To this point, psychological design rationale has allowed our View Matcher work to be deliberate and cumulative. We feel that projecting the rationale for our current system as we design additional reuse scenarios, and as we prototype them, provides a powerfully articulate foundation for principled design argumentation. It allows us greater confidence that we are standing on the shoulders of our prior View Matcher design work, and the psychological claims it embodies, and not merely in the vicinity of prior work. But alas, this confidence suggests a final complicating thought: A View Matcher for reuse clearly embodies specific knowledge and strategy claims for programmers who first used the View Matcher for learning. Thus, as we can better articulate and more deliberately emulate and develop the “View Matcher approach,” we may move on to the psychological consequences of designing species of artifacts.
4. PALEONTOLOGY-AS-YOU-GO

History is an intriguing concept in a design field like human-computer interaction. One reason is that the here and now recedes into the distant past with such astonishing speed. One wonders whether people like Dan Bricklin and Bob Frankston, the inventors of VisiCalc, ever expected to be cultural historians, much less dinosaurs. And yet, whatever else they are, they certainly are both of these — and only a decade after they were young turks! (See the interview in Licklider, 1989.) The history of electronic spreadsheet applications for microcomputers dawned just in the fall of 1979, yet it is already rich and valuable, even a bit hoary, to anyone seeking to understand modern technology or to successfully impact the course of future technological evolution. Bricklin and Frankston saw all of that history, but more than this, they created that history by creating the ancestor of subsequent spreadsheets.

This is another reason why history is so intriguing in HCI: it is quite deliberately, routinely, and profoundly created by HCI designers in the course of their workaday activities. In the case of the spreadsheet, a few simple but powerful insights into how people work, and how they might work, entailed a substantial revolution in how they do work, and, we may be sure, in how they will work in the future. This is weighty stuff: how can we manage our design activities in a satisfying and effective manner when our frames of reference are always radically dynamic and the implications of what we do are oftentimes immediately far-reaching?

Francis Bacon seems to have anticipated that modern science and technology might beget such pressures and responsibilities. Design rationale, taken broadly to incorporate all the approaches described in this special issue and others now being pioneered, can be seen as instantiating his “natural history of trades.” Because design rationale is developed within the design process itself, it mitigates the paradox of instant history by integrating the analysis and abstraction of a design with the creation and implementation of that same design. Design rationale can help us build a pertinent understanding of the context, the users, the tasks, the technologies, the situations — as we go. In the seventeenth century, a lag of 5 or 10 decades in the development of a history of technology turned out to be a decisive limitation, for the spreadsheet even a single decade would have been prohibitive. Indeed, a reasonable projection is that HCI design history needs to be codified instantaneously in order to be useful.

Our particular approach to design rationale has been to treat it as a vehicle for building contextualized science out of practice. The background for us was the observation that situated artifacts serve as embodied theories in the practice HCI, the foreground was the question of how to make this observation useful. Design rationale provides a way of getting the implicit theory out of the artifact and its situation of use, and into a form that is public and explicit. The representations we have exhibited here are only semi-formal but they are more disciplined and precise than the post hoc case study papers we referred to earlier and they entail a design practice that is more structured and accountable than direct emulation of prior art. This is exactly the kind of consequence we are after: we want to support what is evident, successful and (therefore perhaps) natural about current design practice in HCI (the transaction between tasks and artifacts in use, the treatment of artifacts as embodied theory) but we want to augment this practice (though not so abruptly or precipitously as to damage or undermine what makes it attractive and efficacious).

Can we do better? We are keenly aware of how difficult it has been to build an applicable and intellectually significant science base for HCI (Carroll & Campbell, 1986). Our strategy now is to let the design material we work with dictate its own analysis to the extent we can. We have adopted a principle of “ontological minimization” (Carroll, 1990b): the scientific ontology of a practical domain should add as little as necessary to

6 Dauntingly, we must also consider that if these developments are allowed to be merely "evolutionary," they may also always be optimized too locally, e.g., Gould, 1989.
existent practical ontology. This ecological value system urges us to distinguish between those concepts and techniques which are applicable in understanding and creating tasks and artifacts, and those which are not. On these grounds, again, design rationale appears particularly suitable to us: it impels grounding the scientific interests of HCI research in the practical concerns of HCI design.

We can clearly do better. Psychological design rationale already justifies itself to us in our own design work. Our practice with it has converged rapidly in the past two years, and it has also proven to be rich in spawning work on new issues and application domains in our laboratory. For example, we are now applying it to the design of design tools (which has raised interesting issues about designing scenarios in which scenarios are designed — and evaluating claims and tradeoffs at multiple levels of use). We are also exploring how conventional cognitive descriptions of users (e.g. task-action grammars), which have been of very limited direct use in HCI design, might find use as tools for developing psychological design rationale (Payne, 1991).

For us, three sorts of concerns lie in the immediate future: programmatic, methodological and instrumental. As mentioned above, our research program is essentially to exercise and develop our approach through our own design work, and to explore significant theoretical issues within the context of these design exercises. An important issue in the View Matcher example is the generalization and abstraction of the design rationale occasioned by our decision to build a View Matcher for code reuse. This is of course a very limited illustration of what we briefly described as a psychology of tasks (Figure 2). Our work showed how general concepts like paradigmatic examples, view coordination and filtering could be abstracted and generalized from one design project to another.

More significantly, it showed that detailed psychological consequences could be generalized and adduced within a subsequent design argument: The usage vignettes in the View Matcher for reuse are a specific development of the example applications in the View Matcher for learning; both exemplify typical functionality, but the learning examples are interactive in order to engage the user where the reuse examples are scripted in order to minimally disrupt the user’s work. Similarly, the usage episodes in the View Matcher for reuse were a development from the application transactions defined by breakpoints in the View Matcher for learning; both provide decompositions of the example, but the former delineates the typical roles (patterns of messages and behavior) of a particular object where the latter delineates a series of messages sends involving a variety of objects but over a more limited range of function and course of time. This suggests how prior HCI design work can provide a basis for subsequent design, but it also raises many questions. For example, how will the exercise go when there are a thousand design rationales in our inventory instead of one or two? Confronted with this, designers may feel that it’s better just not to know!

At the same time, we are poignantly aware of how small a piece of the design rationale problem we are developing. Our analysis concentrates, roughly speaking, on the cognitive psychology constituted in HCI artifacts. But clearly other domains of analytic discourse should be brought to bear: by purporting to have value, social utility, and internal structure, designed artifacts embody claims pertaining to economics, sociology, software engineering, electrical engineering, and so forth. Indeed, Harrison, Roast and Wright (1989) specifically suggested that our notion of psychological design rationale be complemented by claims pertaining to software engineering and systems design issues in a more comprehensive design rationale. But even granting limitation to the domain of cognitive psychology, one could ask whether the vocabulary of that domain is uniquely appropriate or appropriate at all (to wit, will designers ever bother to penetrate this vocabulary?). It is an open question whether the distinctions and generalizations supported by cognitive psychology compensate for the potential insularity of using its jargon.

Methodologically, we feel that we have demonstrated the feasibility of psychological design rationale: we can do what we wanted to be able to do and it seems to add value. This
is compelling to us in part because our work was directed at a realistic HCI design problem, and in fact has produced one system already in use outside our laboratory. Of course, this is only the first and the smallest step, though it justifies the effort of bothering to worry further. We need to explore the role that psychological design rationale might play in developing analytic summative evaluation in HCI design (evaluation directed at collections of interconnected psychological claims, not at whole artifacts or isolated features of artifacts). We need to investigate the reliability and transferability of our method to other designers and analysts, as we continue to investigate and improve its efficacy (we have had some encouraging preliminary experiences with this, Bellamy & Carroll, 1990). To some extent, we are addressing this by publishing worked examples, concrete models that can be critiqued and adapted. All of our work, however, pertains to exploratory systems design (small groups implementing powerful tools on powerful platforms) in contrast to product design (large groups working within existing and diverse constraints). We have begun to expand our designer-user set by providing courses and conference tutorials. We have a long way to go.

Finally, and directly related to the issue of managing a science base of design rationales, is the instrumental concern — tools. The tabular summaries we used in this paper may be the stuff of research discourse, and may even be useful to researcher-designers like us, but we have no illusion that these presentations could be generally suitable. One approach we imagine is designing a hypermedia browser for exploring claims, artifacts and tasks, as targets and as mutual contexts, presented as text, spoken narration and testimonial, graphics, video episodes, etc. We are only at the very beginning of this effort (cf. Conklin & Begeman, 1988; Fischer, Lemke, McCall, & Morch, 1991 [this issues]).

Francis Bacon’s “natural history of trades” is an intriguing idea. In the electronic welter of HCI, this idea is alive in an exciting and multifaceted zeitgeist directed at realizing the deliberated evolution of useful and usable tools and environments. We believe that this work will transform practice in HCI design by rendering the evolution of new tasks and artifacts more deliberative, even science-based. Beyond this, we believe it may transform our understanding of the nature and role of science in practice, perhaps providing a paradigm for a psychology of design.

Acknowledgements
This work was made possible by the rich context provided by our colleagues. In particular, we thank Rachel Bellamy for collaboration since 1988 on psychological design rationale of Smalltalk (which will be described more fully in her forthcoming Cambridge University Ph.D. dissertation), Rachel Bellamy and Janice Singer for collaboration in 1989 on the initial claims analysis of the View Matcher for learning, Eric Gold for discussion of reifying the usage context of Smalltalk instances, and Christine Sweeney for collaboration in the development of the reuse View Matcher system. Our general framework and its application in this paper owe much to Rachel Bellamy, Robert Campbell, Wendy Kellogg, Steve Payne, and Kevin Singley. We also got important suggestions from John Bowers, Jeff Conklin, Tomasz Ksiezyk, Clayton Lewis, Tom Moran, and Randy Smith.

ORIGINAL PUBLICATION INFORMATION
REFERENCES


