Using Cognitive Dimensions:
Advice from the Trenches

Jason Dagit, Joseph Lawrance, Christoph Neumann,
Margaret Burnett, Ronald Metoyer, Sam Adams

March 12, 2006

Abstract

Many researchers have analyzed visual language design using Cognitive Dimensions (CDs), but some have reinterpreted the purpose, vocabulary, and use of CDs, potentially creating confusion. In particular, those who have used CDs to convince themselves or others that their language is usable have tended to ignore or downplay the tradeoffs inherent in design, resulting in evaluations that provide few insights. Researchers who do not consider *who*, *when*, and *how* best to analyze a visual language using Cognitive Dimensions are likely to miss the most useful opportunities to uncover problems in their visual languages. In this paper, we consider common breakdowns when using Cognitive Dimensions in analysis. Then, using three case studies, we demonstrate how the *who*, *when*, and *how* circumstances under which Cognitive Dimensions are applied impact the gains that can be expected.

1 Introduction

Visual language design is hard. Balancing language soundness, expressiveness, and environment is a delicate act that greatly affects a user's experience with a visual language. There is a temptation to dive into the features of the language and work on the user experience later. Or one may simply be taken by surprise when, after a large visual language development effort, a user study finds a fundamental discrepancy between how users actually behave and how the designer expected users to behave.

Cognitive Dimensions (CDs) [20] provide a broad-brush approach to help a visual language designer consider cognitive implications and tradeoffs of design decisions. The first time the most experienced of us (Burnett) used CDs, which was in 1996, we evaluated two existing visual languages that our collaborators and we had written. We gained immediate insights into issues we had neglected to consider before. We were immediately impressed with this aspect and, ever since that time, we have used CDs to evaluate any new subsystems and languages that we design.

Still, in that first experience, we quickly ran into trouble and we have seen other researchers have the same trouble. The trouble we had was this: what can be accomplished with CDs for evaluating existing visual languages is easily misunderstood. Many researchers do not realize that CDs are limited in the same way that testing is limited. Just as it is not possible to "prove" a visual language is correct with testing, it is not possible to prove a visual language will be acceptably usable with CDs. Rather, it is only possible for these mechanisms to find the *existence* of problems. Yet, people are often tempted to use CDs (and testing) in exactly this way, as though they really can be used to "prove usability" (similarly, testing cannot be used to "prove correctness").

This problem is exacerbated when the visual language being evaluated is one's own visual language nearing completion, because a conflict of interest arises: the designer has a strong incentive to decide that the system has no usability issues. Thus, as soon as the system's designer finds an opportunity to approve some aspect of a CD, he or she may be tempted to pronounce that CD as being satisfactorily considered. Instead, a designer should consider each dimension a little longer before moving on to the next CD.

On the other hand, when the system being evaluated is an emerging system, these conflicts and temptations are diminished. In an emerging system, the designer is highly motivated to find every single problem possible. The motivation to find problems early is natural, so as to quickly address problems rather than having them crop up at a later, more expensive time, such as after implementation.

In this paper, we aim to help future visual language designers in their use of CDs by identifying and describing the common breakdowns in the CDs literature; and showing the circumstances that can lead to or avoid these breakdowns. We begin by considering the uses of CDs reported in the literature, identifying the breakdown in CD usage exhibited in previous work. Then, through three case studies of actual visual language projects in which we used CDs, we examine the successes and failures in light of how three factors—*who* uses CDs, *when* CDs are used, and *how* one uses CDs—affect the ability to find usability problems.

Our work differs from other published accounts of CDs in two primary ways. First, it recounts use of CDs to help with real projects. Most other accounts use CDs as an academic exercise in order to demonstrate concepts, provide examples, or publish "validations" of designs. Second, unlike other accounts, it focuses on lessons learned, breakdowns encountered, and how to avoid such breakdowns in the use of CDs.

2 Brief Background on Cognitive Dimensions

Although this journal issue celebrates ten years of the Cognitive Dimensions framework, Cognitive Dimensions predates Green and Petre's seminal publication in 1996 [20]. In 1989, Green first developed Cognitive Dimensions to analyze the correspondence between a programmer's cognitive strategy and the information structures within a programming language [18]. This early paper established several of the Cognitive Dimensions, but the framework did not take off as an analytical tool until after Green and Petre refined the Cognitive Dimensions framework seven years later, when they applied Cognitive Dimensions to visual languages in their paper [20] and in Green's keynote address at the IEEE Symposium on Visual Languages [19]. These papers established Cognitive Dimensions as a broad-brush approach for considering the psychological aspects and tradeoffs that underly the design of notations and environments. In the time since the original publication, researchers have authored numerous papers citing the Cognitive Dimensions framework, and have applied the Cognitive Dimensions to notational systems other than programming languages. The Cognitive Dimensions are summarized in Table 1.

Table 1: The Cognitive Dimensions [17]

Dimension	Definition
Abstraction gradient	Types and availability of abstraction mechanisms
Closeness of mapping	Closeness of representation to domain
Consistency	Similar semantics are expressed in similar syntactic forms
Diffuseness	Verbosity of language
Error-proneness	Notation invites mistakes
Hard mental operations	High demand on cognitive resources
Hidden dependencies	Important links between entities are not visible
Premature commitment	Constraints on the order of doing things
Progressive evaluation	Work-to-date can be checked at any time
Provisionality	Degree of commitment to actions or marks
Role-expressiveness	The purpose of a component is readily inferred
Secondary notation	Extra information in means other than formal syntax
Viscosity	Resistance to change
Visibility	Ability to view components easily

3 The Cognitive Dimensions and Breakdowns

Research citing the CDs have used the framework in two primary ways. The first way is to apply CDs to visual language designs (e.g., [1, 6, 7, 13, 21, 24, 29, 32]). The second way is to develop new intellectual tools for using CDs (e.g., [3, 9, 10, 12, 16, 30]), so as to provide specific guidance, procedures, or artifacts for CD usage.

In this section, we focus on the key breakdowns we have noticed in the research that applies CDs when analyzing systems, and how the intellectual tools can help prevent these breakdowns.

3.1 How Applying Cognitive Dimensions Can Break Down

For those applying the Cognitive Dimensions framework, CDs have been used both as a vocabulary for describing the cognitive aspects of a visual language's *design* while the design is still emerging, and as an intellectual tool for *evaluating* visual languages and environments. Unfortunately, breakdowns can occur in both types of usage.

3.1.1 Breakdowns in Using Cognitive Dimensions as a Vocabulary

Several researchers have used the vocabulary of the Cognitive Dimensions to describe the needs of users in order to inform design, as was the original intent of the CDs framework. However, when using the vocabulary of CDs, it is necessary to discuss design tradeoffs revealed through the interaction of different dimensions, because changes to a design that attempt to resolve problems in one dimension can often lead to even worse problems in another dimension. Therefore, detailed discussions of only one dimension (as in [28]'s focus on *closeness of mapping*), or brief discussions of each dimension (as in [15]) may not account for design tradeoffs because the discussion is too narrowly focused or too shallow to demonstrate how the CDs interact. We emphasize that these types of breakdowns are not necessarily failures on the parts of the researchers using the vocabulary in this way. Rather, the problem arises when this method of reporting leads to breakdowns by the readers who follow up on these works, because the missing information can lead to false inferences. Even so, reporting all the dimensions is not viable in a paper with limited space, and this paper is one that does not list all the analyses actually performed. We recommend that researchers should take care to emphasize tradeoffs in the way the dimensions interact in their discussions.

An even more critical type of breakdown can occur when researchers apply their new interpretations or definitions to the existing CDs. Since the CDs were intended to be a shared vocabulary, reinterpreting and redefining existing dimensions obscures the original definitions, confusing others using CDs. Some researchers who reinterpret selected dimensions contribute to the body of CDs research by introducing new dimensions as extensions to CDs.[1] However, other researchers who reinterpret selected dimensions effectively redefine existing dimensions in their papers; this is where the breakdown occurs, because multiple conflicting definitions confuse other researchers. For example, in the context of one system, the researchers used "low abstraction gradient" and "simplicity" interchangeably [23], which might have been appropriate for the particular system being considered, but causes misunderstandings for those using the paper as an instructive example in the use of CDs. Blackwell identified this as a growing problem, noting that it is increasingly important to record common misunderstandings of the dimensions [6].

3.1.2 Breakdowns in Using the Cognitive Dimensions for Analysis

Researchers using Cognitive Dimensions can demonstrate that flaws exist through the limitations uncovered by a CDs analysis. However, researchers cannot assert that no flaws exist when they have uncovered no flaws, because absence of evidence is not evidence of absence. Thus, researchers who uncover no flaws in their research using Cognitive Dimensions have essentially wasted their time: *Cognitive Dimensions are not useful as a tool for acceptance*.

Despite this, a number of research papers use Cognitive Dimensions for this very purpose [4, 11, 14, 25, 31]—and for good reason. The reason is that visual languages whose design goals are to benefit people in some way need some form of evaluation, and CDs are more practical in terms of designer time than other alternatives such as empirical user studies. Yet, when evaluating one's own system after it has been extensively implemented over time, it becomes very difficult to see the errors, in part because the designers have accustomed themselves to working around them.

The essence of the breakdown is the use of Cognitive Dimensions for a purpose they cannot accomplish: acceptance instead of finding usability issues. Because of the conflict of interest, we do not advocate waiting until this late to use the CDs, especially on one's own system. That said, using CDs late on one's own system is better than not using them at all. The third case study in this paper will demonstrate reasons for both of these opposing arguments.

3.2 Extensions and Tools for the Cognitive Dimensions

Researchers have developed several intellectual tools to bring measurement devices, concreteness, or cross-cutting ideas to the process of applying CDs, and use of these tools may help to avoid some of the breakdowns. Prior to the original publication

[1]Extensions to the CDs framework [21, 24] are beyond the scope of this paper, although Blackwell has discussed several proposed extensions and criteria for selecting extensions [6].

establishing the Cognitive Dimensions framework, Payne and Green developed Task-Action Grammars (or TAG) to measure the consistency of a notation [29]. Yang et al. developed Representation Design Benchmarks to measure some of the Cognitive Dimensions for static representations [32]. Hundhausen, et al. developed an experimental framework for analyzing the closeness of mapping dimension [22]. Blackwell developed a questionnaire to enable users to evaluate notations and environments using Cognitive Dimensions [7, 8]. Blackwell also developed an attention investment model as a unifying analytical approach to CDs [5]. Understanding that Cognitive Dimensions represent tradeoffs in design that affect different types of users in varying ways, Clarke developed Personas to analyze the fit between a profile of users and a system based on the Cognitive Dimensions framework [13]. Throughout this paper, we provide examples of using several of these intellectual tools.

4 Case Study #1: Applying CDs Early — Interactive Football Playbook

Cognitive Dimensions may be applied very early in the development of a visual programming system—during the design phase before any development is done. Our group of visual language and computer animation researchers are developing a football simulation environment, the Interactive Football Playbook (IFP), which allows American football coaches to program and visualize football plays. To understand how coaches represent and communicate plays and strategies, we observed football coaches instructing players in the classroom and we interviewed football coaches. Using this information, we started designing the IFP.

We were highly motivated to find design problems early in the design process. Our goal was to look for usability problems at a high level before committing the design into source code, so we applied a CDs analysis on a paper prototype (Figure 1) using the CDs Questionnaire [7].

Who: The original designers of the IFP.
When: The inception of design.
How: Green and Petre's CDs paper [20] and CDs Questionnaire [7].

4.1 Background on the IFP

The Interactive Football Playbook allows American football coaches to create digital playbooks with animated content. Such an environment requires the presentation of a large amount of information. This information can be associated with virtual agents, spatial locations, particular play formations, etc. The language is modeled after the notation coaches already use in their playbooks and classroom demonstrations. The language is still under development.

Coaches program the virtual players through a constraint-based, visual programming language and execute the simulation while visualizing the execution in both 2D and 3D. Coaches create offensive and defensive formations and then combine formations and rules to create scenarios. Rules are associated with players to define the players' behavior. Ultimately the rules will be a mix of pictographic and textual representations. The pictographic portion of a rule parallels the coach's notation and the textual portion allows the coach to parameterize the rule or denote rules with no pic-

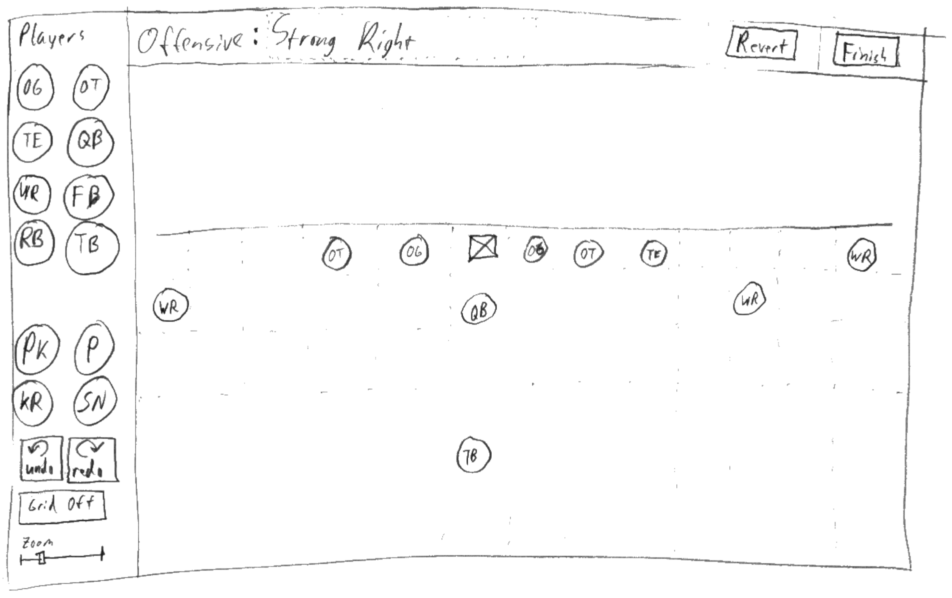


Figure 1: A portion of the paper prototype: the offensive player formation editing screen for the Interactive Football Playbook. The user would manipulate a player by dragging it from the toolbox, around the field, or off the field.

tographic component. Execution is performed by unifying the rules and sending the results to the animation engine.

4.2 Analysis Process

We created a mockup for the offensive formation editing screen (see Figure 1.) A brief written description accompanied the mockup to describe how one interacts with the interface. For example: the players are placed and moved through dragging, the interface is zoomed by dragging the slider, etc. We used the CDs Questionnaire [7] to perform a formal Cognitive Dimensions analysis on the mockup to find problems with the design.

The Cognitive Dimensions Questionnaire [7] is a general-purpose tool to enable end users (as well as designers) to apply the Cognitive Dimensions framework to systems. The CDs framework alone is not easily applied by end users since one must be familiar with the framework to apply its concepts and vocabulary. By framing the concepts of CDs into questions, the questionnaire is intended to make the framework accessible to end users for use in evaluating a system.

Although the CDs Questionnaire targets end users, we (as system designers) turned to the questionnaire because it provided us with the advantage of concrete questions to illuminate the abstract descriptions of the dimensions in [20]. Because the questions provide a perspective on Cognitive Dimensions, the questionnaire caused us to consider aspects of the IFP interface beyond our own interpretation of CDs. This served to increase the breadth of our analysis.

We performed our analysis with the assumption that American football coaches will be our end users. We consider it a problem when the environment imposes an unnecessary or inappropriate cognitive burden on this particular audience. The CDs framework allowed us to analyze the cognitive aspect of our environment in an abstract way (independent of specific users), but we achieved our strongest results when we took into account the tasks and characteristics of American football coaches.

4.3 Insights and Design Maneuvers

Our Cognitive Dimensions analysis identified a number of problems and design considerations. When using CDs, attempting to improve in one dimension often affects other dimensions. These changes are referred to as design maneuvers, and a good discussion of them is found in [17]—including a nice graph of common relationships for tradeoffs in the dimensions. Below, we will summarize our results and identify possible design maneuvers and their implications. Evaluating these tradeoffs is done by thinking carefully about the design goals, in our case, the design goals for the IFP with respect to football coaches (our target audience).

4.3.1 Visibility, Hard Mental Operations and Abstraction

A design goal in the IFP is to provide football coaches with high visibility for all parts of a formation, because in the classroom football coaches draw and discuss football plays in their totality. We supported this design goal by allowing the coach to work with an overview of the formation and then zoom into parts of the formation. Even with this in mind from the beginning, we still found problems with our design by using the CDs Questionnaire.

For example, below are the questions with our answers from the Visibility/Juxtaposition section of the CDs Questionnaire:

Question 1: How easy is it to see or find the various parts of the notation while it is being created or changed? Why?

Answer: The user can find all the parts of the notation while being changed because all the players are on the screen simultaneously. A problem occurs when zooming in: players can go off the screen and there is no indication that they are “out there.”

Question 2: What kind of things are more difficult to see or find?

Answer: It is a little difficult to find types of players since there are so many.

Question 3: If you need to compare or combine different parts, can you see them at the same time? If not, why not?

Answer: The user cannot compare two formations at the same time. The user can toggle between the two easily, but cannot have them on-screen at the same time. Also, the user cannot view an offensive formation in relation to a defensive formation.

Likewise, we found problems in the other dimensions by answering the questions for the other dimensions in a similar manner.

Answering the questions served to identify problems, but not until we started to develop remedies did we begin to encounter tradeoffs. For example, the zooming problem, identified from question one, requires the user’s working memory to retain which players have been placed but are not visible (a hard mental operation). One remedy might be to incorporate a bird's eye view into the UI so the user can see where the zoomed area is in relation to other players. This would increase the visibility of the system and reduce the mental operations, however it introduces new notations which constitute the interface for the bird's eye view. These notations, although they do not increase the abstraction gradient of the main notation, are abstractions which the user must learn and understand. However, the learning curve may not be significant if the interface is consistent with notation the user already understands.

The problem identified from question three is that the user cannot compare two formations at the same time. The user may toggle between two formations easily, but cannot have more than one formation on-screen at the same time. It became clear in our classroom observations that football coaches need to be able to compare two different formations of the same kind as well as pair an offensive formation with a defensive formation to see the alignment. This creates another hard mental operation by taxing the user's working memory since the user must compare a visible formation with another formation stored in their working memory.

In order to support viewing arbitrary formations at the same time, we could allow a user to float a formation in its own window. By floating two formations, they could be compared and edited simultaneously. Though this change allows for juxtapositioning of formations, the change introduces a tradeoff: coaches must learn a new concept, floating a window, and its associated interface (an abstraction representing that concept). Likewise, allowing coaches to see the alignment of two opposing formations could be supported by allowing a defensive formation to be displayed on the layout field while the offensive formation is being edited. Once again, this would require the user to learn the mechanism (another abstraction) for displaying the opposing formation.

4.3.2 Hidden Dependencies, Premature Commitment, Error-proneness and Viscosity

Formations are the basis of scenarios, and as such, are abstractions themselves. This creates a more powerful notation than having to create formations of players in each scenario since formations can be defined once and reused. Our design, while seeking to increase expressive power, falls victim to hidden dependencies, premature commitment, error-proneness and high viscosity.

A formation may be modified once and affect all scenarios built on that formation. Since formations are edited separately from scenarios, when a formation changes, there is no indication scenarios will be changed and of the extent of the change; this creates a hidden dependency for the user. If there are many scenarios depending on a certain formation, then the effect of the change can be very large. This also contributes to error-proneness within the IFP since the system leads the user to make changes that could introduce errors.

Because formations are the basis of scenarios, the user is forced to think first in terms of formations and then in terms of combining formations into scenarios. This approach can cause a problem if the coaches are developing a new formation in response to a certain scenario. This creates a premature commitment since the coach must decide on an arrangement of players before the scenario is created and may not move players when editing the scenario. Allowing the user to view an opposing formation while editing partially solves this problem since it provides an alignment mechanism, but that solution still does not allow the coaches to adjust a formation within a scenario.

Our choice to abstract formations from scenarios also creates viscosity problems. The formation editor itself demonstrates low viscosity—one simply drags a player around to modify the formation—but the inability to change formations within scenarios creates a more viscous notation for scenarios. We could attempt to lower viscosity by allowing the user to move players within the scenario editing environment, but that raises the question of the effect of such a change. When a formation is modified within a scenario, it could either update the formation in all scenarios (globally) or only in the current scenario (locally). A global change creates an especially dangerous hidden dependency because the environment seems to suggest changes to one scenario will not affect others (there is no visible indication of change), but a local change weakens the power of the abstraction—editing a formation via the formation editor may not update all the places where that formation is used.

Our CDs analysis brought out these design tradeoffs: (1) abstracting out formations gives the notation more power, but creates a hidden dependency; (2) requiring the user to create a formation before a scenario creates premature commitment, but removing this requirement prevents us from abstracting out formations (3) allowing a user to modify a formation within the scenario editor makes the notation less viscous, but creates a hidden dependency; and (4) allowing the user to customize a formation within the scenario editor makes the notation less viscous but weakens the power of the abstraction. We resolved (1) and (2) by considering our design goals, but (3) and (4) will require more observations or interviews to find out to what extent the coaches will create or adjust new formations within a scenario.

4.3.3 Provisionality and Progressive Evaluation

The formation editor exhibits high provisionality in the sense that the user may edit the scenario in an ad-hoc manner without having to commit to a player arragement until the user determines the formation to be complete. The user may "play" with the formation to sketch out an idea. The user can re-arrange players at any time and see the result. However, the formation editor does not allow the user to make formations less precise. To support less precision, the environment could incorporate notation to indicate that a player is positioned in an approximate location instead of an exact one. Once again, this increases the abstraction gradient by introducing a new notation. By making the less-precise notation optional, the user would not be forced to use (or even learn) the new notation.

There is a sense of progressive evaluation as a formation is constructed since there is clear visual progress as the players are positioned in the formation. However, the user may not incorporate partial formations into a scenario. So, while the user may create a partial set of rules within a scenario and evaluate them, the user cannot extend this idea by having partial sets of players. Removing this barrier would allow coaches to progressively develop formations within an associated context of a scenario. That change would also serve to increase the provisionality of the system since the user would not have to create formations first and then scenarios.

Extending provisionality and progressive evaluation in the IFP may allow for more opportunistic development, but it could result in hidden dependencies between the formations and scenarios. From our interviews and classroom observations, it is not clear if coaches need to develop formations in an ad-hoc manner. In the classroom, we have primarily observed coaches working from a canon of formations, but the CDs analysis pointed out to us that we need to observe the coaches doing new formation development as well, to obtain the necessary insights into the formation development process.

4.3.4 Whoops, No Secondary Notation!

The notation in the IFP is modeled after the notation coaches already use in their playbooks, providing good closeness of mapping. This source also leads to good role-expressiveness, since the formation in the IFP looks like a formation in the coach's printed playbook. This representation also exhibits low diffuseness (is compact) since there is one symbol for one player and each kind of player takes the same amount of space to represent. And, although there are different kinds of players, the notation maintains consistency by having players retain the same shape and use different labels. Thus, these particular CDs make the design look very good.

An important advantage in CD analysis is that CDs provide a specific list of concepts to consider, so that important concepts do not slip by without being considered. In our case, this is exactly what happened. The CDs pointed out to us that there is a complete absence of secondary notation within the system. We do not provide an escape from formalism: a way for one to make notes to oneself or add informal information within the environment. In real life, coaches write notes all throughout their printed playbooks, so the environment should support some form of commenting or augmentation attached to different players, formations, and scenarios. For example, this would allow a coach to include the goals and tasks for a player in a particular formation.

4.4 IFP Summary

The CDs framework enabled us to identify a number of problems and tradeoffs very early in the design process for the IFP. Identifying the tradeoffs early on helped guide our design process and steer us away from problems that would inhibit our design goals. We used the CDs questionnaire to provide us with a concrete list of things to check and consider about our design. This approach allowed us to find things we had failed to consider at all. In particular, it was our experience that the questionnaire achieved its intended purpose of being accessible: even those in our group who had low levels of familiarity with CDs were able to identify problems with the IFP by answering the questionnaire.

5 Case Study #2: Applying CDs Mid-Cycle — Adieu

Many visual languages are designed iteratively, with later versions refining earlier ones. Cognitive Dimensions can be extremely useful between two of these versions, both to reveal problems not already realized and to help avoid introducing new problems when attempting to solve known existing ones.

Case Study #2 occurred at exactly this stage. Our group performing the analysis consisted of visual language researchers and designers working together on the next version of the language. Some of the group had been part of the original design team, and some had not.

Because a language version already existed, we had an existing implementation that we viewed as a prototype of the next version, and this provided an extremely useful platform for our analysis, since all details of the language at this stage were known and accessible (unlike Case Study #1, which was so early in the design, many of the details had not yet been worked out). We were highly motivated to find flaws with the existing design, because we wanted to prevent as many flaws as possible from being present in the version being planned.

In this project, we employed a number of approaches to using CDs. We chose Task-Action Grammars because they allowed us to focus specifically on consistency flaws at the exclusion of other dimensions. After the consistency analysis we stepped back and analyzed our new design by considering all the dimensions with respect to our original goals, as described by design refactoring in Section 5.2. This case study will recount the ways we used these devices. In addition, we used the CDs Questionnaire [7], but do not report that aspect here, since it has already been discussed in Case Study #1.

Who: A group of researchers and designers, some of whom had been on the original design team and some of whom had not.
When: Between versions of the language. (Current Adieu and Future Adieu)
How: Green and Petre's CDs paper [20], design refactoring, and Task-Action Grammars [29].

Here the *who*, *when*, and *how* differ from Case Study #1 in that we are not the original designers, we are applying CDs between versions and we are using tools which are more targeted.

The visual language we analyzed was the Ad-Hoc Development and Integration Tool for End Users, or ADIEU [2], which was created by one of the authors (Adams) with collaboration at IBM. The language's goal is to enable end users to develop web services and web applications without any Java or J2EE knowledge. Users develop these applications by using collections of "cards," each of which act like single-function applications in a form-based, desktop-like environment. The data fields in cards can be used like cells in a spreadsheet and can contain either data or an expression that determines the data at run time. Cards can also run other cards; this capability provides the basic flow control necessary for programming concepts such as decision branching, sequences, and repetition.

5.1 Issues Revealed for Current Adieu

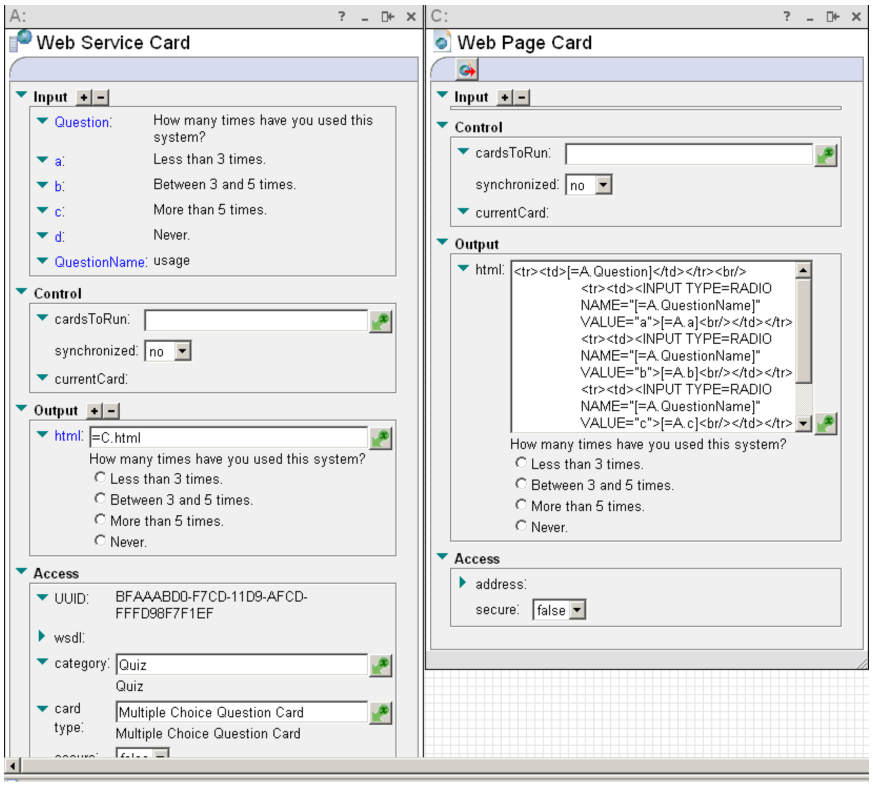
5.1.1 Consistency

Originally (a few versions back), Adieu had only two types of loop, one that executes while an expression is true and one that makes use of a counter. Experience with that version revealed commons patterns of loop usage, leading to the creation of two more loop constructs: "list iteration" and "list iteration with concatenation" of intermediate results. The latter two loop constructs take a list as a parameter and execute a card for each element of the list. Thus, at the time of our evaluation, there had come to be four types of iteration.

We turned to Task-Action Grammars [29] for more precise analysis of the consistency issues. Appendix A shows the details of our TAG analysis.

5.1.2 Error-proneness and Hard Mental Operations

A major source of errors relates to references and automatic naming of cards. Cards are created and assigned names in alphabetic order starting with "A". The user cannot change card names as used in references, although the user can add a phrase "naming" a card. In a scenario where the user needs to change the type (or functionality) of a card in an existing Adieu program, the user has two options. The user can delete the card



```
A:
Web Service Card
Input
  Question:      How many times have you used this
                 system?
  a:             Less than 3 times.
  b:             Between 3 and 5 times.
  c:             More than 5 times.
  d:             Never.
  QuestionName:  usage
Control
  cardsToRun:
  synchronized:  no
  currentCard:
Output
  html:  =C.html
         How many times have you used this system?
          ○ Less than 3 times.
          ○ Between 3 and 5 times.
          ○ More than 5 times.
          ○ Never.
Access
  UUID:      BFAAABD0-F7CD-11D9-AFCD-
             FFFD98F7F1EF
  wsdl:
  category:  Quiz
             Quiz
  card type: Multiple Choice Question Card
             Multiple Choice Question Card
```

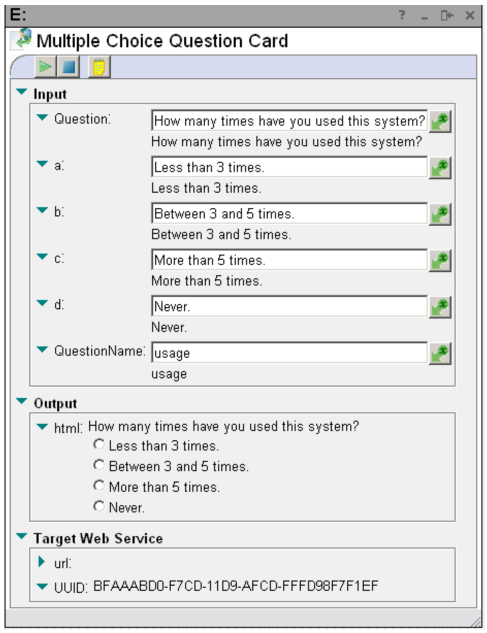
```
C:
Web Page Card
Input
Control
  cardsToRun:
  synchronized:  no
  currentCard:
Output
  html:  <tr><td>[=A.Question]</td></tr><br/>
                 <tr><td><INPUT TYPE=RADIO
                 NAME="[=A.QuestionName]"
                 VALUE="a">[=A.a]<br/></td></tr>
                 <tr><td><INPUT TYPE=RADIO
                 NAME="[=A.QuestionName]"
                 VALUE="b">[=A.b]<br/></td></tr>
                 <tr><td><INPUT TYPE=RADIO
                 NAME="[=A.QuestionName]"
                 VALUE="c">[=A.c]<br/></td></tr>
         How many times have you used this system?
          ○ Less than 3 times.
          ○ Between 3 and 5 times.
          ○ More than 5 times.
          ○ Never.
Access
  address:
  secure:  false
```

Figure 2: Example Adieu program showing a user defined web service "Quiz Card." The card on the left provides the programatic interface while the card on the right contains the logic of the "Quiz Card." Informally, the card on the left is the function signature and the card on the right is the function body.

that is already there and create the new card in its place, somehow remembering how to fill in the new card based on the old one. Or, the user can create a new card, copying over the relevant parts from the old card and remembering to update any references to the old card so that they refer to the new card.

Hard mental operations are also present when reading Current Adieu programs. Understanding the relationship between cards and fields in a sheet is difficult. In some cases we have found it important to draw a sketch on paper of how the parts fit together. An important redesign for Adieu is to include relationship visualizations. Future Adieu will include arrows for many of the relationships such as field and card references.

References are not the only source of error-proneness and hard mental operations in Current Adieu. When checking the formula language for consistency we discovered that lists of card identifiers were expected in the form: "A,B,C,D". The formula language has another type of list which is general purpose and works with any of the built-in list formula functions. The syntax of the general purpose lists requires that the list be wrapped in curly braces, as in the example list "{1,2,3}". The CDs analysis


E:

Multiple Choice Question Card

Input

Question: How many times have you used this system?
How many times have you used this system?

a: Less than 3 times.
Less than 3 times.

b: Between 3 and 5 times.
Between 3 and 5 times.

c: More than 5 times.
More than 5 times.

d: Never.
Never.

QuestionName: usage
usage

Output

html: How many times have you used this system?
○ Less than 3 times.
○ Between 3 and 5 times.
○ More than 5 times.
○ Never.

Target Web Service

url:

UUID: BFAAABD0-F7CD-11D9-AFCD-FFFD98F7F1EF

Figure 3: This Adieu program uses the "Quiz Card" web service pictured in Figure 2. The example cards show how an interactive quiz for a website might be constructed using Current Adieu.

pointed out that having these two distinct notations was both error-prone and inconsistent.

Our proposed change to the list syntax is to use only one list notation where commas are used to separate list elements. When a list is used inside of an expression, parentheses may be used to signify precedence and to separate the list from the surrounding expression(s).

5.1.3 Hidden Dependencies and Viscosity

Current Adieu has several types of dependencies each based on field references. A field may reference another field, a card or a web service. When a field references another field or a card, both must reside within the same program, which is known as a *sheet* in Adieu terminology. If a field references a web service the referenced web service could be an Adieu program or a third-party web service that is available on another server, possibly over the Internet. The first two hidden dependencies are very similar to those found in spreadsheet languages. A hidden dependency that arises from a web service presents an interesting challenge. The user may not have control over the availability or behavior of the web service.

The current implementation only shows dependencies at the card level. One of the goals of Adieu is that mistakes should be easy to find and fix. To help Adieu meet this goal it was important for us to recognize these hidden dependencies. For example, we discovered that web service cards create hidden dependencies. Web service cards are designed to create a level of abstraction and thus hide certain details. The failing in our language is that we do not support the user in seeing dependencies of the web service's interface. For example, a program could be written that uses a web service and if that web service is changed it could break the existing program.

In Adieu, problems related to hidden dependencies are worsened by viscosity issues. Showing dependencies is important due to the viscosity related to changing a card's type. Once a card has been created the functionality is fixed by the type of card. Changing the behavior of a card requires replacing the card by a new one. The overall effect is that changing the behavior in parts of an Adieu program, for example changing the type of loop construct, can be very viscous. On the other hand, making most formula and parameter changes is very easy. To help deal with the viscosity of changing card type we proposed the card transformation wizard in Section 5.2.2.

5.1.4 Closeness of Mapping

The target audience for Adieu consists of users who are already familiar with using a web browser and interacting with web pages. The main notation of Adieu is a card (a form). The notation is meant to be reasonably close to forms found on web pages. The expected Adieu audience will have experience with spreadsheets, and Current Adieu takes advantage of this by using field expressions and card references in much the same way as spreadsheets. Although cards are not meant to reflect spreadsheets — and hence closeness of mapping is not quite the right way to describe this concept — it was this dimension that started us thinking concretely about the audience and what they could be expected to know.

In considering this issue, we were satisfied with the connection to spreadsheet notation. However, this was not the case with the HTML notation (refer back to Figure 2 Web Page Card on the left). Explicit, textual, HTML does not provide closeness of mapping to the problem domain. We realized that without a WYSIWYG HTML editor the current means of generating web pages and web content will not be close enough to the end result for our goals.

5.1.5 Secondary Notation

Every dimension adds new insights. Although we had not originally expected secondary notation to be of much use to us, given our particular domain and audience, this dimension turned out to draw our attention to important flaws that we would otherwise have overlooked.

Regarding Current Adieu's secondary notation, cards can be given descriptive non-semantic names. These descriptive names may be arbitrary, allowing the user to essentially label each card. Current Adieu also allows the user to place cards anywhere on the screen. Thus, the user may use layout to convey meaning.

These examples at first convinced us that secondary notation was already adequate, but our CDs analysis led us to realize that Future Adieu could benefit from other forms of secondary notation. For example, we propose "description cards" which have no language semantics, but simply carry a description or may be attached to relationships between cards and fields. This change will also allow the user to annotate dependencies within Adieu programs, potentially reducing problems with hidden dependencies and hard mental operations. Paying close attention to secondary notation gave us a chance to improve Future Adieu in ways we had not expected.

5.2 Design Refactoring and Design Maneuvers

In the same sense that software engineers refactor source code [27], applying CDs between versions of a visual language allows the designers to "refactor" the design. In software engineering terms, refactoring code is the process of preserving external behavior while transforming the internal structure of the source code[2]. Software engineers value code refactoring as evidenced by the availability of tools for refactoring. For large software projects, code refactoring brings robustness, consistency and maintainability to the source code [26].

Here, we use the term *design refactoring* to mean changing the language design while preserving the functionality of the language. For example, in design refactoring a designer might change syntactic constructs for looping while preserving loop semantics. When refactoring a visual language design, the goal is to provide a better experience for the user of the language.

In the Current Adieu language, the design had evolved over several iterations and conflicted with itself in places. During the iterations, new features were added to deal with certain use cases. We used Cognitive Dimensions to guide our restructuring. For example, the dimension of consistency was very important in this process to help us find flaws that had grown into the design.

To solve the consistency problems with loops described earlier, we devised a combined loop card with the help of Task-Action Grammars (refer back to refapp:tag). Figure 4 shows the combined loop card for Future Adieu. Below is our CDs analysis of this design maneuver.

5.2.1 Design Maneuver #1: Combined Loop Card

Our combined loop card design maneuver was inspired by our Task-Action Grammar analysis. For details about the analysis see Box 2. Below is our CDs analysis of this design maneuver.

Viscosity and error-proneness With the combined loop card it is possible to change looping details without a need to fix stale card references that may be introduced by creating a new card that makes an existing loop card obsolete.

Role-expressiveness Future Adieu's combined loop card replaces the four loop constructs in Current Adieu. Since the only name needed in Future Adieu is "Loop Card," the need for a user to differentiate among the roles of four different types of loops is eliminated.

Premature commitment In Current Adieu the type of loop must be selected before the details may be edited. If the user knows that a loop is needed he or she may not know if the loop should terminate on a boolean expression or a counter value. By combining the looping constructs in Future Adieu, changing between types of loops requires changing the loop type *on* the card.

Here we see trade offs between consistency and error-proneness. While this new loop structure scores highly in the consistency dimension it may be more error-prone for users who are not familiar with different forms of loops. Also, the new looping structure can be very intimidating in terms of the many options that the user must

[2]The website http://www.refactoring.com/ provides a portal to information on the topic of refactoring.

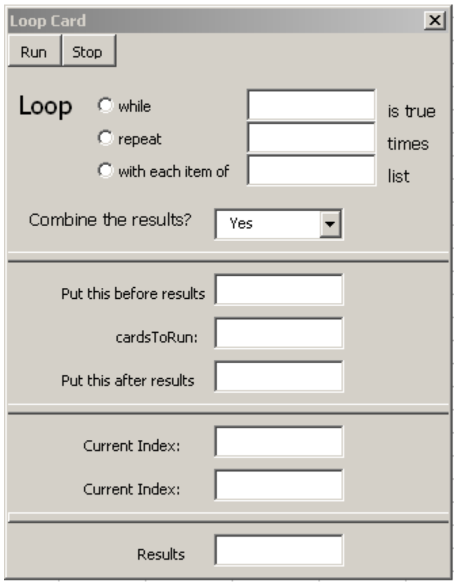


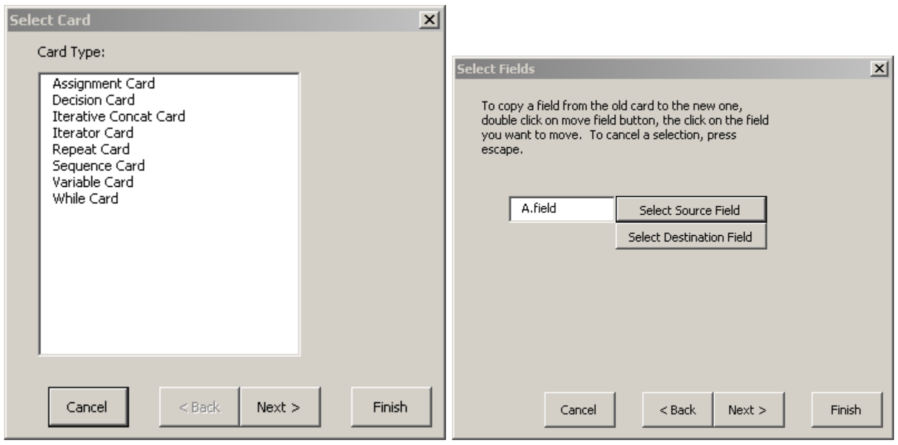
Figure 4: Combined loop card prototype for Future Adieu. This card combines the loop constructs found in Current Adieu into one card.

go through to use it. We also realized that other cards and common constructs in the language would benefit from a reduction in viscosity and error-proneness when changing card types.

We decided to keep looking for a better alternative instead of adding a combined loop card. Eventually we thought of the card transformation wizard, explained below, which is a more general solution to this problem. One of the strong advantages to the card transformation approach is that we provide one way to transform between cards instead of an ad-hoc way on a per language construct basis.

5.2.2 Design Maneuver #2: Card Transformation Wizard

The purpose of the card transformation wizard is to assist the user in changing a specific card which already exists in a Future Adieu program to another type of card. The wizard is invoked via any card in a Future Adieu program. The first step asks the user to choose a new type of card. The next step works by allowing the user to select a source field on an existing card, whose content they may alter, and then allowing the user to select a destination field in the new card, which was chosen in the first step of the wizard. The user is allowed to repeat the selection process as desired until all the fields that need to be transfered have been transfered. A prototype of the wizard is given in Figure 5. To help prevent an increase in hard mental operations it is important to provide a visualization of the relationship between fields in the new card and old card. The wizard removes the original card and replaces it with the new card. The new card's name is taken from the original card that the user used to invoke the wizard. Since fields may change the wizard automates the process of fixing dependencies within the Adieu



To copy a field from the old card to the new one, double click on move field button, the click on the field you want to move. To cancel a selection, press escape.

(a) Step One (b) Step Two

Figure 5: Card transformation wizard prototype steps one and two. The card transformation wizard allows the user to select fields that should be moved from an existing card to a new card.

program.

Consistency By having a special wizard which the users need to invoke to transform cards, we are increasing the number of concepts the user needs to learn which cannot be inferred from the rest of the language. This could potentially be viewed as a consistency flaw. On the other hand, adding one wizard to the language which allows transformation between any two types of cards is much more consistent than providing several multi-function cards. The combined loop card approach described above is an example of providing a multi-function card.

Viscosity and error-proneness By automating the process of changing from one card to another we are reducing the risk of errors introduced by requiring a human to find and change all relevant field and card references.

Premature commitment Adding the wizard does not change the level of premature commitment, but the cost to recover from choosing the wrong type of card is greatly reduced.

A comparison with Section 5.2.1 shows that the card transformation wizard has the same advantages in error-proneness, viscosity and premature commitment as the combined loop card. The extra advantage for the card transformation wizard comes in the consistency dimension. With the combined loop card, we (the designers of Future Adieu) would need to identify all possible cases where a card transformation would be useful and, if we succeed at at that, devise a new language feature for each such case. All of these patched-on language features would need to be kept consistent in some sense. Since the card transformation wizard provides a general solution that allows the user to use a single mechanism to transform between existing or even arbitrary cards in Future Adieu, it avoids inconsistencies that could arise among multiple new features, as well as the inconsistency that could arise if some transformation case was not anticipated by the designers.

5.3 Adieu Summary

By using the CDs framework between versions of the language we were able to refactor the language design. Specific tools, such as Task-Action Grammars, provided insight into specific dimensions. Using CDs and tools based on CDs led us to design maneuvers we may have never considered otherwise. After the focused analysis and corresponding changes to the language we were able to step back and look at the language again, as a whole, using the CDs framework.

6 Case Study #3: Applying CDs as a Summative Evaluation — SkinBuilder

As mentioned in the introduction, there is an inherent conflict of interest in doing a summative (end of the day) evaluation on one's own system in order to broadcast its flaws to the world. When we evaluated SkinBuilder — a product that we did not develop — the company behind the product was planning on a release date only a few months out. As we discovered, applying CDs to an existing product has different characteristics than applying CDs in formative evaluations. In particular, the motivation to find usability problems at a later stage is greatly affected by *who* does the CDs evaluation.

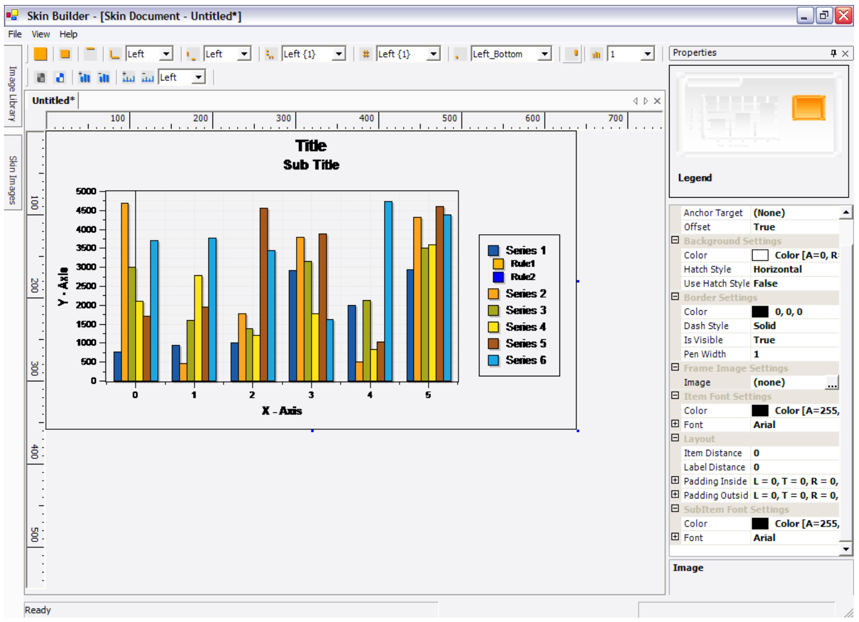
Who: Visual language researchers who were not the developers of SkinBuilder.
When: Just months before SkinBuilder's estimated ship date.
How: CDs original paper.

In the previous two case studies, we developed or collaborated with the developers of the visual languages under evaluation at an earlier stage in development. In this case study, we worked as external reviewers for a finished product.

From a brief demo of the product, we learned that SkinBuilder (shown in Figure 6) was a WYSIWYG tool for customizing the look of software-generated charts. The company behind SkinBuilder and the corresponding chart-generation software envisioned graphic designers at software and web development firms as the target audience for the product. Although the company behind SkinBuilder had previously involved users to inform other product designs, for this new, yet unreleased product, the company had no users to guide SkinBuilder's design. Thus, the company did not evaluate their product with user studies, nor did the company evaluate their product using CDs.

After we learned the purpose of the SkinBuilder, we evaluated the software using the Cognitive Dimensions framework.[3] Soon after the demo, we acquired an evaluation copy of the software that we could use independently on our own machines. By evaluating SkinBuilder on our own machines without assistance from the developers, we could avoid influence from the developers. Once we acquired the software, we evaluated the system for several hours. We set out to reproduce what the developers had demonstrated to us, and we took lengthy notes on what we encountered during our evaluation.

[3]Cognitive Dimensions can describe the usability of any notational system, including including computer-aided design tools like SkinBuilder [6].



Anchor Target	(None)
Offset	True
⊟ Background Settings	
Color	⬜ Color [A=0, R
Hatch Style	Horizontal
Use Hatch Style	False
⊟ Border Settings	
Color	⬛ 0, 0, 0
Dash Style	Solid
Is Visible	True
Pen Width	1
⊟ Frame Image Settings	
Image	(none)
⊟ Item Font Settings	
Color	⬛ Color [A=255,
⊞ Font	Arial
⊟ Layout	
Item Distance	0
Label Distance	0
⊞ Padding Inside	L = 0, T = 0, R = 0,
⊞ Padding Outsid	L = 0, T = 0, R = 0,
⊟ SubItem Font Settings	
Color	⬛ Color [A=255,
⊞ Font	Arial

Image

Figure 6: In SkinBuilder, the interface to select a chart element is placed to the left of the manipulation interface and the picture that highlights the current selection.

During our evaluation, we ran into trouble as we attempted to reproduce the demo. We were unable to discover several of the features demonstrated earlier, so we scheduled time after our initial evaluation to meet with the developers to learn more about the features with which we struggled.

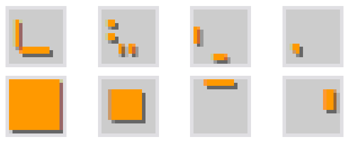
6.1 CDs Analysis

From our evaluation and our meeting with the developers afterward, we noticed that our experience, expectations, and motivations to find problems in the product differed greatly from those of the developers. Obviously, the perspective of the evaluators should influence the CDs evaluation, and this was certainly true in the case of SkinBuilder. We believe the fact that our perspective was so different from the developers' that it contributed greatly to the number of issues we were able to find in SkinBuilder via the CDs — issues that had not been noticed by the developers themselves in their months of working on the product.

6.1.1 Expertise, Role-expressiveness and Visibility

Nowhere was our lack of experience more apparent than in the *role-expressiveness* of the toolbar buttons the developers provided in their interface. According to [20], the dimension of role-expressiveness is intended to describe how easy it is to answer the question "what is this bit for?" From our novice experience, the toolbar buttons (shown in Table 2) were indistinguishable rectangles. Even when we interacted with the toolbar buttons, we could not understand how each button corresponded with the actions we observed. Not until after the developers explained to us how each toolbar button selected particular elements in the chart could we understand their rationale for designing the toolbar buttons as they did.

Table 2: The toolbar buttons in SkinBuilder are difficult to distinguish and convey little meaning to novice users.



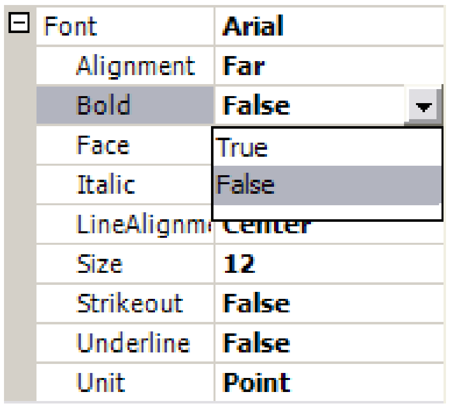
Our inexperience with SkinBuilder highlighted another design choice the developers made which impacted *visibility*. In our evaluation, when we clicked on elements in the chart, such as the title, SkinBuilder did not highlight the selection, or so we thought. As we discovered from meeting the developers after the evaluation, the interface for selecting elements in a chart was placed *away* from the interface for highlighting what was selected.[4] The selected object was highlighted in the upper right corner of the interface (Figure 6), away from the chart itself. Because the two interfaces were not placed together, when selecting an element, a user would need to focus on one side of the screen to select an element and then look at the right side of the screen to determine if the correct element was selected. Thus, users would need to split their focus when selecting chart elements, an essential task in SkinBuilder. The interface for manipulating selected elements, also on the right side of the screen, suffered from the same problem.

6.1.2 Expectations, Closeness of Mapping and Consistency

As we evaluated SkinBuilder, we found that our expectations of how the product would work differed substantially from those of the developers. We expected SkinBuilder would behave like Excel or similar chart-editing software, whereas they had modeled SkinBuilder's user interface after the integrated development environment (IDE) that they used to develop it! Consequently, we were often surprised by the design of the interface, which contributed greatly to our CDs analysis of *closeness of mapping* and *consistency*.

SkinBuilder provided an unusual interface to modify font properties which offered little *closeness of mapping* for end users. Rather than using a standard font chooser, the developers of SkinBuilder displayed properties such as bold as choices between True or False (Figure 7). Although programmers may think of "bold" as a boolean property, we were not convinced that end users would think to set a bold font this way. That is, the *mapping* may be close from the perspective of programmers, but it is not as likely to be a notation similar to end users' existing conventions. Personas, a technique developed to represent the workstyles of a target audience, could have potentially addressed this issue.

[4]In contrast, similar chart-editing products (such as Excel) place a border around the current selection, solving the *visibility* problem.



Font	Arial
Alignment	Far
Bold	False
Face	
Italic	
LineAlignm	Center
Size	12
Strikeout	False
Underline	False
Unit	Point

Figure 7: The unusual font selection mechanism in SkinBuilder maps poorly to an end-user mental model of font selection.

We also expected that we should be able to accomplish the same task in a *consistent* way. Unfortunately, we discovered that we could not select colors the same way throughout SkinBuilder. In some cases, SkinBuilder required us to enter colors as RGB triplets, whereas in other cases, SkinBuilder offered a short list of predefined colors. In other cases, SkinBuilder provided a color palette to select colors. We were not convinced that the mapping between RGB triplets and colors would be obvious for end users.

6.1.3 Motivations and Error-proneness

Users will make mistakes. Additionally, allowing users to recover easily from mistakes encourages them to try new things by reducing the risk of making mistakes. Unfortunately, SkinBuilder did not include an "undo" feature, which makes it difficult to recover from mistakes. The developers themselves had not anticipated this as a problem with their interface. When we pointed out this problem, they told us that although they felt "undo" was important, they were unwilling to bear the large development costs necessary to implement that feature in time for their anticipated release.

This demonstrates the importance of perspective. By the time we evaluated SkinBuilder, our motivation to find usability problems differed greatly from those of the developers. Though the company behind SkinBuilder did not evaluate their own product using CDs, from their demos and our conversations with the company, they tried to demonstrate that their product was ready to ship. On the other hand, we came into the CDs evaluation with the motivation to demonstrate areas of improvement in their product.

6.2 SkinBuilder epilogue

We have not attempted to be complete in this section: we found additional problems using CDs that are not discussed here, but these only belabor the points that we make, so we have highlighted only a few CDs to give an idea of the sorts of issues that were revealed.

Applying CDs in a late evaluation has advantages and disadvantages. Unfortunately, by the time we evaluated SkinBuilder using CDs, it would have been too late for the developers of SkinBuilder to make changes to the product (such as implementing "undo") in time to meet their shipping schedule. On the other hand, after we presented our results to the team behind SkinBuilder, the developers decided to shelve the project. As a conseuqnece, our evaluation prevented the company behind SkinBuilder from throwing good money after bad by taking SkinBuilder to market or to user testing. Recall that when we evaluated their system, they emphasized that SkinBuilder was ready to ship "within months." After our evaluation, they came to realize that SkinBuilder had "never made it out of the prototyping phase." Applying a CDs analysis late in the development process cannot benefit developers as much as applying CDs early in the process, but it can inform developers when to stop work on a project with numerous, serious usability issues.

7 Discussion and Summary

As we have stressed in this paper, the tradeoffs revealed by the CDs are very important. The designer must take care to balance all of the dimensions. Similarly, there is a balance between reporting all dimensions and just the tradeoffs and priorities illuminated by the CDs. Here we have tried to follow our own advice of emphasizing the tradeoffs and priorities.

As the case studies reported in this paper of actual visual language projects' use of CDs demonstrate, *who* applies CDs, *when* they are applied, and the interaction between these two factors, can have a big impact on the extent of CDs' usefulness. In Case Study #1, the analysis was effective largely because of its *who* and *when* factors, because in that combination (early, by the original designer) the language designer was highly motivated to find all the flaws so as to create the best language he could. Case Study #2 also demonstrated this interplay: the *when* was early enough — at the outset of a new version, at which time identifying features to change did not reflect badly on the designers. Yet, it was late enough that the presence of an implementation (the previous version's) allowed detailed investigations that were not possible in Case Study #1. Finally, the inclusion of designers not on the original design team allowed fresh insights beyond those the original designers had noticed.

Case Study #3 showed that a completed visual environment can also gain from a summative CDs analysis, but not in the same way as CD analyses done earlier in a system's design. In this summative use of CDs, the main benefit was to reveal the significance of the design flaws, thereby preventing the additional expenditure of resources on a product unlikely to succeed in the marketplace. Because we as the evaluators were an outside party, we avoided potential conflicts of interest which can weaken the motivation and ability to find usability problems.

Particularly in summative CD analyses, it is important to consider *who* is evaluating the system: the developers themselves or a third party. When evaluating their own system, developers eager to ship a product have a strong incentive to decide that the system has no usability issues. Furthermore, developers know too much about how their product works and how much time they spent implementing particular features in the language; at this point, it may be quite difficult for them to see the perspective of an end user. For this reason, if CDs are used late in development the designer is tempted to use CDs to validate the design instead of using them to improve the design.

The essence of the breakdown is that Cognitive Dimensions cannot validate usability; rather, they can only find usability problems.

The *how* factor can help avoid this breakdown: the misuse of CDs to attempt to validate usability. The *how* factor can also prevent other breakdowns, such as the misunderstanding of CDs' definitions or incomplete evaluations. For example, Personas and TAGs help the designer to maintain objectivity. They do this by providing a designer with a way of measuring one or more of the CDs. In the case of Personas, they also tie in a strong connection to some segment of the intended audience. The CDs Questionnaire helps avoid breakdowns too, in that it makes concrete the meaning of each CD by requiring answers to specific questions. This helps to prevent both misunderstandings and incompleteness.

Of these three factors, the *when* factor is probably the most critical in delivering value from CDs. The CDs analysis in Case Study #1, which was early in the design, pointed out numerous usability problems before time and money were spent on development. Case Study #2 achieved some of the same benefits — it was not as available for design tinkering as Case Study #1, but still was at a good point in its lifecycle to make significant changes, and its greater maturity allowed more valuable detailed analyses to be performed. Both of these case studies are in sharp contrast to Case Study #3. With SkinBuilder in Case Study #3, the CDs analysis found so many problems that the company developing the product decided to abandon it due to the high cost of changing the design so late in development. Although it is tempting to view Case Study #3 as a failure, it is important to note that the CDs analysis saved the company from spending more money on a product unsuitable for the intended audience.

In summary, the case studies reported in this paper demonstrate three main points: (1) it is never too late to use CDs to analyze a design, but the earlier, the better; (2) the CDs can be used by either the original designer or external evaluators, but this *who* factor interacts with the *when* factor; and (3) use of some of the specific intellectual tools that have been devised for CDs, such as the CDs Questionnaire, Task-Action Grammars, or Personas, can help with the CD analysis because they can contribute a stronger connection to the intended audience, concrete step-by-step guidance, and measuring devices. Our primary advice from the trenches is this: Designers of a language should use CDs early (earlier in design) and use CDs often (iterative design and design refactoring). But if CDs are used late in the process they should be used by external evaluators. Although using CDs is never a bad idea, the *who*, the *when* and the *how* determine how much value a user of CDs can get from them.

8 Acknowledgments

This work was supported in part by NSF CCR-0237706 and by the EUSES Consortium via NSF ITR-0325273. Jason Dagit's work was also supported by an IBM internship, and Margaret Burnett's work was supported by an IBM International Faculty Award.

APPENDIX

A Task-Action Grammars

In [20], consistency is stated as "when a person knows some of the language structure, how much of the rest can be successfully guessed?" To help analyze consistency we used Task-Action Grammars [29] (TAG). This tool was developed by Payne and Green for analyzing consistency of command languages and user interfaces. Task-Action Grammars are grammars that represent a task-oriented model of how an efficient user would use the interface or command language.

A.1 Intended Use

Language designers construct a grammar that represents the way an interface could be used by an efficient user. Each production in the resulting grammar represents one concept the user is expected to know when using the system. By comparing TAGs of similar interfaces, designers can obtain a concrete, although approximate, measure of consistency. This is because Task-Action Grammars with fewer productions represent fewer concepts that must be mastered or learned and thus a more consistent interface [29]. We used Task-Action Grammars to obtain a quantification of the consistency of Current Adieu compared with Future Adieu. (We caution that finding a language design that can be described via a very simple TAG is not a reason to stop looking for good designs.)

A.2 How Task Action Grammars Work

We performed a TAG analysis of Adieu. Below is a TAG which represents looping constructs in Adieu. The TAG on the left represents the four independent loop cards and the TAG on the right represents the combined loop card. We have deliberately rendered the productions unreadable, so as to emphasize the aspect of interest: namely, the *number* of productions has been greatly reduced in the TAG on the right. (One production is given in readable form at the bottom.)

$$
\begin{aligned}
Task[Effect = \text{looping}, Type] &\rightarrow cardPallete[Card = \text{value-from-production}, Type = \text{value-from-production}]\\
&+ fillInCard[Card, Type]
\end{aligned}
$$

$$
\begin{aligned}
fillInCard[Card, Type = \text{simple}] &\rightarrow fillInSimpleLoopCard[Card]\\
fillInCard[Card, Type = \text{simple-special-case}] &\rightarrow fillInSpecialLoopCard[Card]\\
fillInCard[Card, Type = \text{complex}] &\rightarrow fillInComplexLoopCard[Card]
\end{aligned}
$$

$$
\begin{aligned}
fillInSimpleLoopCard[Card] &\rightarrow selectField[field1 = \text{value-from-production}, Card]\\
&+ editField[field1 \doteq \text{cardsToRun}]\\
&+ selectSynchronization\\
&+ selectField[field2 = \text{value-from-production}, Card]\\
&+ editField[field2]
\end{aligned}
$$

$$
\begin{aligned}
fillInSpecialLoopCard[Card] &\rightarrow selectField[field1 = \text{value-from-production}, Card]\\
&+ editField[field1 \doteq \text{list}]\\
&+ selectField[field2 = \text{value-from-production}, Card]\\
&+ editField[field2 \doteq \text{cardsToRun}]\\
&+ selectSynchronization
\end{aligned}
$$

$$
\begin{aligned}
fillInComplexLoopCard[Card = \text{IteratorConcat}] &\rightarrow selectField[field1 = \text{value-from-production}, Card]\\
&+ editField[field1 \doteq \text{list}]\\
&+ selectField[field2 = \text{value-from-production}, Card]\\
&+ editField[field1 \doteq \text{cardsToRun}]\\
&+ selectSynchronization\\
&+ selectField[field3 = \text{value-from-production}, Card]\\
&+ editField[field3 \doteq \text{header}]\\
&+ selectField[field4 = \text{value-from-production}, Card]\\
&+ editField[field4 \doteq \text{body}]\\
&+ selectField[field5 = \text{value-from-production}, Card]\\
&+ editField[field5 \doteq \text{footer}]
\end{aligned}
$$

(a) Before: Separate loop constructs

$$
\begin{aligned}
Task[Effect = \text{looping}] &\rightarrow cardPallete[Card = \text{value-from-production}]\\
&+ fillInCard[Card]
\end{aligned}
$$

$$
\begin{aligned}
fillInCard[Card = \text{looping}] &\rightarrow selecType[Type = \text{value-from-production}]\\
&+ selectField[field1 = \text{value-from-production}, Card]\\
&+ editField[field1]\\
&+ combineResults[SaveResults = \text{value-from-production}]\\
&+ editLoopBody[Card, SaveResults]
\end{aligned}
$$

$$
\begin{aligned}
editLoopBody[Card, SaveResults = \text{no}] &\rightarrow selectField[field1 = \text{value-from-production}, Card]\\
&+ editField[field1 \doteq \text{cardsToRun}]
\end{aligned}
$$

$$
\begin{aligned}
editLoopBody[Card, SaveResults = \text{yes}] &\rightarrow selectField[field1 = \text{value-from-production}, Card]\\
&+ editField[field1 \doteq \text{"put this before results"}]\\
&+ selectField[field2 = \text{value-from-production}, Card]\\
&+ editField[field2 \doteq \text{cardsToRun}]\\
&+ selectField[field3 = \text{value-from-production}, Card]\\
&+ editField[field3 \doteq \text{"put this after results"}]
\end{aligned}
$$

(b) After: Combined loop construct

$$
\begin{aligned}
fillInCard[Card = \text{looping}] &\rightarrow selecType[Type = \text{from-production}]\\
&+ selectField[field1 = \text{from-production}, Card]\\
&+ editField[field1]\\
&+ combineResult[SaveResults = \text{from-production}]\\
&+ editLoopBody[Card, SaveResults]
\end{aligned}
$$

Example TAG Production

References

[1] R. Abraham, M. Erwig, S. Kollmansberger, and E. Seifert. Visual specifications of correct spreadsheets. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 189–196, 2005.

[2] S. S. Adams. Adieu: Ad hoc development and integration tool for end users, August 2005. `http://www.alphaworks.ibm.com/tech/adieu`.

[3] L. Beckwith, M. Burnett, and C. Cook. Reasoning about many-to-many requirement relationships in spreadsheets. In *HCC '02: Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02)*, page 149, Washington, DC, USA, 2002. IEEE Computer Society.

[4] S. Best and P. Cox. Programming an autonomous robot controller by demonstration using artificial neural networks. In *IEEE Symposium on Visual Languages and Human Centric Computing*, pages 157–159, 2004.

[5] A. Blackwell. First steps in programming: A rationale for attention investment models. In *IEEE Symposium on Human Centric Computing Languages and Environments*, pages 2–10, 2002.

[6] A. Blackwell, C. Britton, A. Cox, T. Green, C. Gurr, G. Kadoda, M. Kutar, M. Loomes, C. Nehaniv, M. Petre, C. Roast, C. Roes, A. Wong, and R. Young. *Cognitive Technology 2001 (LNAI 2117)*, chapter Cognitive Dimensions of Notations: Design Tools for Cognitive Technology, pages 325–341. Springer-Verlag, 2001.

[7] A. Blackwell and T. Green. A cognitive dimensions questionnaire optimised for users. In A. Blackwell and E. Bilotta, editors, *Proc. PPIG 12*, pages 137–152, 2000. `http://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/CDquestionnaire.pdf`.

[8] A. F. Blackwell and E. Bilotta. A cognitive dimensions questionnaire optimized for users. In A. F. Blackwell and E. Bilotta, editors, *Proc. PPIG 12*, pages 137–154, 2000.

[9] M. Burnett, N. Cao, M. Arredondo-Castro, and J. Atwood. End-user programming of time as an 'ordinary' dimension in grid-oriented visual programming languages. *Journal of Visual Languages and Computing*, 13(4):421–447, 2002.

[10] M. Burnett, S. K. Chekka, and R. Pandey. Far: An end-user language to support cottage e-services. In *HCC '01: Proceedings of the IEEE 2001 Symposia on Human Centric Computing Languages and Environments (HCC'01)*, pages 195–202, Washington, DC, USA, 2001. IEEE Computer Society.

[11] Q. Chen, J. Grundy, and J. Hosking. An e-whiteboard application to support early design-stage sketching of uml diagrams. In *IEEE Symposium on Human Centric Computing Languages and Environments*, pages 219–226, 2003.

[12] S. Clarke. Evaluating a new programming language. In G. Kadoda, editor, *Proc. PPIG 13*, pages 275–289, 2001.

[13] S. Clarke. Measuring API usability. Dr. Dobb's Journal Special Windows/.NET Supplement, May 2004.

[14] L. Ford. Cognitive dimensions of PrologSpace. In *Proceedings of 8th Annual Workshop of the Psychology of Programming Interest Group PPIG-96*, 1996.

[15] S. Gauvin and T. Smedley. Reduction of cognitive load through the addition of high-level semantics to ReactoGraph. In *IEEE Symposium on Visual Languages and Human Centric Computing*, pages 181–188, 2004.

[16] J. Good. VPLs and novice program comprehension: How do different languages compare? In *IEEE Symposium on Visual Languages*, pages 262–269, 1999.

[17] T. Green and A. Blackwell. Cognitive dimensions of information artefacts: A tutorial, Oct 1998. Version 1.2.

[18] T. R. G. Green. Cognitive dimensions of notations. In A. Sutcliffe and L. Macaulay, editors, *People and Computers V*, pages 443–460, Cambridge, UK, 1989. Cambridge University Press.

[19] T. R. G. Green. The visual vision and human cognition. In W. Citrin and M. Burnett, editors, *IEEE Symposium on Visual Languages*, Boulder, Colorado, 1996. Invited talk.

[20] T. R. G. Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *J. Visual Languages and Computing*, 7(2):131–174, Jun 1996.

[21] C. Hundhausen. Using end user visualization environments to mediate conversations: A 'communicative dimensions' framework. *Journal of Visual Languages and Computing*, 16(3):153–185, 2005.

[22] C. Hundhausen, R. Vatrapu, and J. Wingstrom. End-user programming as translation: An experimental framework and study. In *IEEE Symposium on Human Centric Computing Languages and Environments*, pages 47–49, 2003.

[23] C. H. Kim, J. Hosking, and J. Grundy. A suite of visual languages for statistical survey specification. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 19–26, 2005.

[24] A. Ko and B. Myers. Development and evaluation of a model of programming errors. In *IEEE Symposium on Human Centric Computing Languages and Environments*, pages 7–14, 2003.

[25] Y. Li, J. Grundy, R. Amor, and J. Hosking. A data mapping specification environment using a concrete business form-based metaphor. In *IEEE Symposium on Human Centric Computing Languages and Environments*, pages 158–166, 2002.

[26] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[27] W. F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, Champaign, IL, USA, 1992.

[28] J. Pane, B. Myers, and L. Miller. Using HCI techniques to design a more usable programming system. In *IEEE Symposium on Human Centric Computing Languages and Environments*, pages 198–206, 2002.

[29] S. J. Payne and T. R. G. Green. Task-action grammars: A model of the mental representation of task languages. *Human-Computer Interaction*, 2:93–133, 1986.

[30] S. Peyton-Jones, A. Blackwell, and M. Burnett. A user-centred approach to functions in Excel. In *International Conference on Functional Programming*, pages 165–176, New York, NY, USA, 2003. ACM Press.

[31] A. Winn and T. Smedley. Multimedia workshop: Exploring the benefits of a visual scripting language. In *IEEE Symposium on Visual Languages*, pages 280–287, 1998.

[32] S. Yang, M. M. Burnett, E. DeKoven, and M. Zloof. Representation design benchmarks: A design-time aid for VPL navigable static representations. *J. Vis. Lang. Comput.*, 8(5/6):563–599, 1997.