# Deceived by Ease of Use

## Using Paradigmatic Applications to Build Visual Design Environments

**Kurt Schneider, Alex ander Repenning**

Department of Computer Science and Center for LifeLong Learning and Design
{ktschnei, ralex}@cs.colorado.edu
University of Colorado at Boulder
Campus Box 430, Boulder, CO 80309-0430

## Abstract

*Application frameworks for visual design environments usually offer a wide range of features and easy-to-use mechanisms to develop applications. We observed that sometimes those features deceive application designers: Tempted by the desire to make rapid progress, designers go into too much detail about easy things too early in the process, like graphical representations. After the easy-to-use mechanisms have been exploited, they find themselves stuck and frustrated. Premature design decisions made during the feature-driven phase can corrupt application system architecture or require abandonment of much work. Extensive rework endangers project success.*

*Paradigmatic applications can help to bridge the gap between application framework features and intended application – better than manuals or additional features can. As examples and sources for reusable components, this special kind of exemplary applicationsdirects the attention of designers to higher-level building blocks and helps them to avoid premature feature exploitation. We characterize paradigmatic applications and describe their impact on the design process.*

**Keywords:**

Application framework, visual design environment, analogies, examples, design process

## Introduction

In recent years, applications such as graphical notation editors, computer-aided design and engineering environments with interactive graphical interfaces, visual programming languages, and customized direct manipulation tools have gained increasing popularity.

The demand for these types of applications can no longer be met by building them from scratch. Application frameworks promise to facilitate the design and implementation of interactive graphical systems. In an application framework, the common components and features of a class of applications are factored out and made available within a generic application. Class libraries are provided to specialize this generic framework to meet the specific needs of a concrete application. In this paper, we subsume the above class of interactive graphical systems under the notion of visual design environments.

We carried out and supervised about 50 application-building efforts using two different, independently developed application frameworks for visual design environments (Agentsheets: Repenning, 1995; and vis-A-vis: Lichter and Schneider, 1993). While most applications fit well into the generic architectural frames, every now and then a more complicated application seemed to suffer from the need to map its structure to the generic architecture inherited from the framework. Alarmingly, application designers (sometimes including ourselves) initially were excited about the good support provided by the application framework and the rapid progress made. At some point, progress stagnated dramatically and problems occurred. Finally, those applications ended up with really messy designs, or were canceled.

We found that in some breakdown cases, the application design process had been driven by features of the framework. As the analyses of several breakdowns (such as the two examples discussed below) showed, the frameworks had in some cases encouraged designers to get started without deliberately designing their applications. Both example frameworks afforded the design of sophisticated graphical representations as well as operations that were impressing for the application designers (such as bending icons, as in Fig. 3, or making the appearance depend on the internal state of the semantic object). However, the seemingly harmless process of designing a visually appealing graphical representation had implications on the internal structure of the semantic model. Designers refused to revise those structures, instead trying to maintain even premature design results. As a consequence, more serious problems arose, which had to be fixed and patched. Designers and users were discouraged by the unexpected problems and the extreme slow-down in progress. The trap had been set by three trends that were sensible on their own:

- Framework designers obviously wanted to maximize the set of widely useful, basic features. Framework features are presented to be as usable as possible.
- Common features of application frameworks that can be used in a wide range of visual design environments are often located on a very detailed, low-level of abstraction. There is a significant cognitive distance between these features and any complex, concrete application.
- Application designers were driven by their sense of completion; Green (1989) calls such a strategy "opportunistic design." Rapid progress could be made by just doing what was obviously easy to do: exploit the low-level, easy-to-use framework features.

In combination, these trends manifest in a process that deceives application designers. The common ground of visual design environments is the realm of graphical representations. Their creation, modification, and connection to underlying semantic entities are crucial. It is only natural to make those operations easy to use. In this context, however, opportunistic design can be guided by too low a level of design abstraction, and lead to commitments that corrupt higher-level structures.

Our solution to this problem is twofold, reflecting the two roles involved in its creation:

- Framework designers need to provide compelling paradigmatic applications – as comprehensive examples and for reuse of higher-level building blocks. They must illustrate the mapping between the framework and a concrete application.
- Application designers must withstand the extremely strong temptation to indulge into easy-to-use features too early in the process.

These efforts complement each other. We consider the choice of appropriate examples to be far more crucial than their number. While the merit of good examples is widely accepted, there is a lack of an operational characterization of a "good" example. Instead of providing many similar examples that again mainly illustrate the use of framework features, we suggest developing higher-level mechanisms and designing building-blocks and embeding them in "paradigmatic applications" (as defined below).

Section 2, defines the scope of our arguments. Agentsheets and vis-A-vis will briefly be described to illustrate visual design environments. Within this paper, they represent a wide range of similar systems. Section 3 investigates the design process currently followed by (too) many application designers. Section 4, takes a close look at the gap between frameworks and concrete applications. Section 5 provides a characterization of paradigmatic applications to tighten the gap. Section 6, points out how to turn application design from a feature-driven to a more risk-driven approach by reversing the process direction. Instead of encouraging designers to work from the framework to the application, we suggest picking up the application designers close to their applications, guiding them to reusable middle-layer mechanisms, and thus helping them to fill the remaining gap. The final section relates our suggestions to the work of others and concludes our arguments.

## 2 Application Frameworks for Visual Design Environments

Object-oriented ***application frameworks*** contain a generic application and a set of class libraries. The generic application provides an architecture with some empty slots on the detailed level. Common operations of an entire class of applications are implemented on the generic level. A concrete application is constructed by inheriting architecture and common operations from the generic application. Classes from the accompanying class libraries can be used to customize aspects of the generic application. Customization by setting parameters and by inheritance fills the empty slots of the generic application and results in a concrete application. General application frameworks such as ET++ (Gamma *et al.*, 1989) may provide more than one generic application. Sometimes, application frameworks are called "substrates" (McWhirter and Nutt, 1994).

In this paper, we talk about application frameworks built around a generic visual design environment. Visual design environments use a graphical representation on a canvas to work with an underlying (design) model. We refer to the underlying model as ***semantic model***. To afford effective design work in a direct manipulation style (Shneiderman, 1983), the semantic model and representation have to be coupled. The existence and support of a semantic model distinguishes visual design environments from graphical editors.

Common operations of visual design environments provided by application frameworks typically include

- drawing design language symbols, usually by combining graphical primitives such as rectangles, lines, circles and icons;
- editing bitmaps and icons, and representing relationships;
- configuring representations and semantic correspondents to allow parallel creation, modification, and removal of the semantic element and representation;

- allowing interaction with a semantic entity by pointing at its representation and invoking a command;
- performing operations on entire models (such as save, load, print), which are usually implemented in the generic application and inherited by specific applications.

Many frameworks also provide superclasses for semantic elements. Semantic objects can be handled by the framework only when they inherit some "infrastructural" methods from those superclasses. Providing semantic superclasses allows support of common semantic operations, but makes it more difficult to build visual interfaces for existing semantic models.

vis-A-vis and Agentsheets (Fig. 1) are visual design environments. Their appearance is dominated by a canvas or worksheet for the design model. A palette (vis-A-vis) or gallery (Agentsheets) contains icons representing design components. On the canvas, the semantic objects are represented by this same icon (Agentsheets) or by a more elaborate graphical representation (vis-A-vis). Both frameworks provide generic operations on entire models, and allow the definition of application-specific operations on single model elements. In vis-A-vis, element operations are invoked by clicking on the respective element, and then choosing from the popup-menu that appears (see Fig. 1). In Agentsheets, clicks and double clicks are distinguished; custom operations can be invoked by first selecting them from the left-hand side mode bar, and then clicking on the element (e.g., a switch in Fig. 1). vis-A-vis further allows to define customized operations on entire models; they appear in the menu bar above the canvas (Derive Situation Editor, Annotations, Views). Both frameworks allow application designers to access their respective underlying general-purpose programming language, Lisp for Agentsheets and Smalltalk 80 for vis-A-vis.

Despite many similarities, there are major differences between the two environments:

- Agentsheets is grid based. The worksheet is invisibly partitioned in grid cells containing icons (light bulbs, wires, switches). Agentsheets requires semantic elements to be instances of Agent or one of its subclasses (e.g., Position-Aware-Agent, Linkable-Agent, or a custom-made subclass). From those classes, they inherit functionality for "autonomous behavior." They can move from one cell to another, change their displayed icon or produce sounds. Each Agent can easily address its neighbor cells. Relationships usually are implict. They are derived from spatial adjacency of Agents.
- vis-A-vis interprets the canvas as topological space; element positions are meaningless. Relationships are explicitly established and shown. vis-A-vis offers mechanisms to easily address entire models, or all instances of one semantic element class. This allows for simulation, evaluation, code generation, etc. While Agentsheets representations consist entirely of bitmap icons, vis-A-vis allows a combination of a variety of graphical primitives (rectangles, circles, text) to customize symbols. Each component of a symbol can independently react to changes of aspects in semantic element state, and can change style, color, width, etc., accordingly. Entities in Figure 1 combine label text with an automatically fitting rectangle.

vis-A-vis and Agentsheets are used as examples in this paper because they are typical application frameworks for visual design environments. Our arguments apply to a wide range of similar frameworks (Golin *et al.*, 1992; McIntyre and Glinert, 1992).

Some features of an application framework are particularly easy to find and can be used almost without preparation. For example, editing icons are necessary in most application frameworks for visual design environments, and therefore they are usually very well supported and easy to perform.

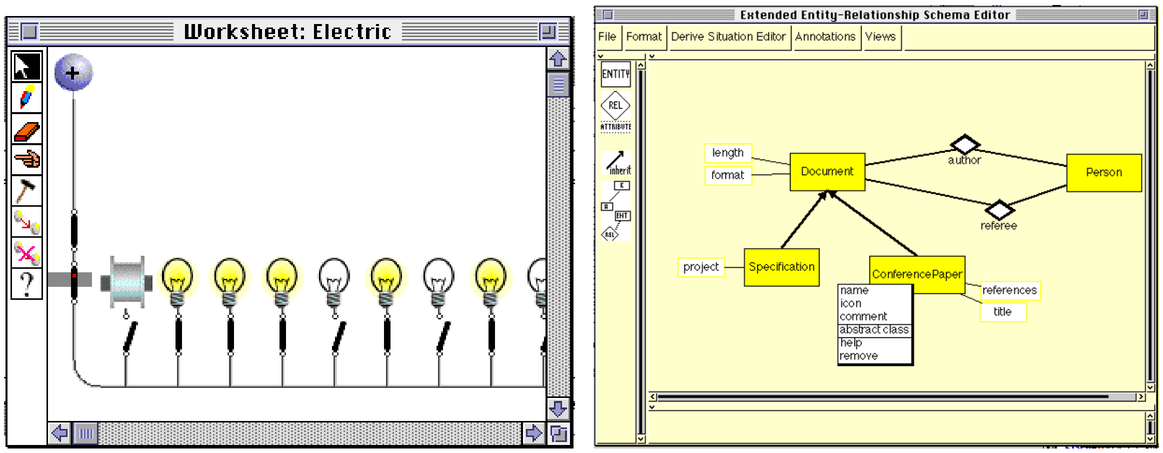


**Figure 1: Typical Agentsheets (left) and vis-A-vis (right) Applications**

A ***framework designer*** is a person who builds an application framework. An ***application designer*** uses a framework to build an application. As a significant part of an application is inherited from the generic application provided in the framework, application design sometimes seems to be trivial and unnecessary. We stick to the notion of application *design*, as we are most concerned about nontrivial applications in which a significant, application-specific part of the system requires further design decisions. We do not assume that application and framework designers work together, or even that they are the same people. If the roles overlap, this may reduce the danger of being deceived, but will not eliminate it.

We call an application designer ***deceived by ease of use*** when the exploitation of easy-to-use features of the application framework has led to an undesirable situation that could have been avoided had those features not been used at that early point in time.

## 3 A Feature-Driven Design Process

When actions and directions are driven by easy-to-use features, the designer starts at the framework and tries to find a way to the application. As the framework exists, but the application does not, the process goes from the known to the unknown.

The further the process proceeds

- the less guidance can be given by the framework, but
- the more work has already been expended and the more code already exists.

Both trends more and more discourage starting anew when a breakdown occurs. Exploiting easy-to-use features *early* defers breakdowns and reinforces these trends.

### Example 1: SESAMsheet

SESAMsheet is an Agentsheets application project that developed a very promising interface within a relatively short time (Fig. 2). SESAMsheet is supposed to visualize information flow in a software development project. Basically, SESAMsheet is a visually enhanced short-cut variant of the SESAM software engineering educational game (Ludewig *et al.*, 1992). Simulated employees and documents can be placed and dragged with the mouse; when simulation time goes by (clock in the upper left corner is activated), simulated employees and documents exchange information. Talking employees open and close their mouths, and utter some conversational sound.

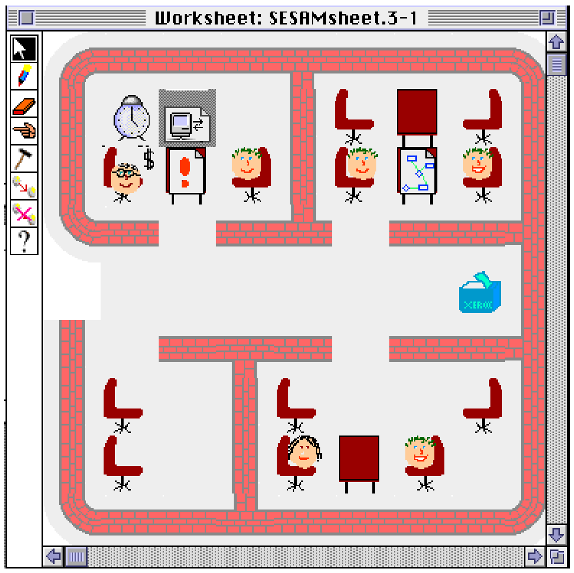


**Figure 2: SESAMsheet: Modeling Information Flow in a Software Development Project**

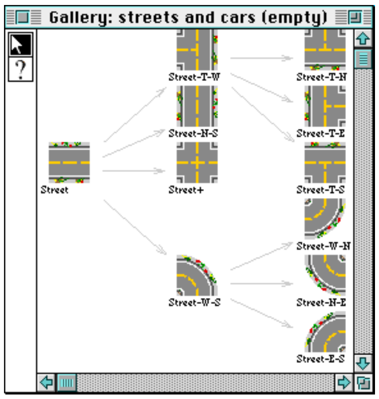


**Figure 3: Bending Icons in Agentsheets**

Most of the state represented by Figure 2 is based on specific Agentsheets features that are very convenient to use. Drawing and placing of icons, definition of reactions to mouse-dragging, and the definition of autonomous behavior are very easy to perform ("talking" is represented by changing between open-mouthed and closed-mouthed icons, together with appropriate sounds). The application designer had the comfortable feeling of making good progress.

Some work went into the perfection of icons: for example, both male and female employees had to be represented. A special feature of Agentsheets, inheritance and bending of icons, further absorbed much attention: The walls around the office consist of icons fitting into the 40x40 pixel grid of this application. As Figure 3 shows, Agentsheets can automatically *bend icons* Repenning (1994) to derive corner-pieces from one given simple piece of walls, roads, or any other icon. This feature is very helpful to save drawing effort; it requires some attention, however, to draw the initial piece exactly symmetric, so that bent and flipped wall pieces neatly fit. One can spend much time making the walls look natural. The impressing interface seems to fairly reward this effort. Like all the other visual design environments shown in this paper, SESAMsheet is a colorful, interactive application.

The SESAMsheet project stagnated when this state was reached; the next step would have been to evaluate the information gathered by all simulated employees, compare it to a given information reference ("the customer"), and send the result back to the employees. Unexpectedly, there was no broadcast or collection mechanism. It was easy for employees to address their adjacent grid cells, but very difficult to exchange information with any central unit. It turned out that Agents (i.e., employees) could easily be linked with each other. The designer of this application now tried to bypass the (missing) broadcast mechanism by explicitly linking employees with the clock. The clock represented the central notion of time. However, this was more difficult than expected because the semantic class of Employees had been defined as a subclass of Position-Aware-Agent to implement talks across a table, not just with immediate neighbors. Linking agents, however, required Linkable-Agent as a superclass, which is in a different branch of the class hierarchy than Position-Aware-Agent. The application designer started to re-implement the link mechanism.

## Example 2: CityPlanner

In one vis-A-vis application, the canvas was supposed to represent Euclidean space (Fig. 4). After scanning a part of a city map as background, planners wanted to place different kinds of colored rectangles on the map, thus indicating different uses of city blocks. Uses, such as commercial, industrial, or residential are color-coded on the pieces. Diagonally striped rectangles represent portions of the block that were expecting change of use. The width of the diagonal line represents the time frame for this expected change (within 2, 5, 10 years). A circle on other pieces represents a given partial use. CityPlanner is based on Arias and Anselin (1982).

vis-A-vis allows the creation of diagonally striped rectangles with changing colors and stripe widths. Each aspect may depend on an aspect of the internal state of the corresponding semantic model element. Problems occurred, however, when an evaluation operation was added that included checking adjacent uses. vis-A-vis does not support the notion of spatial relationships. The designer decided to introduce explicit relationships ("north-of," "south-of," etc.) and make their representation a line of zero width. This idea is in good accordance with vis-A-vis features, as it is easy to introduce relationships. Unfortunately, a very inefficient update mechanism is required to keep explicit "pseudo-spatial" relationships in synchronization with the relative positions of model elements when they are moved. Only later did the designer think of reusing an invisible grid that was provided by vis-A-vis, and used to align symbols.

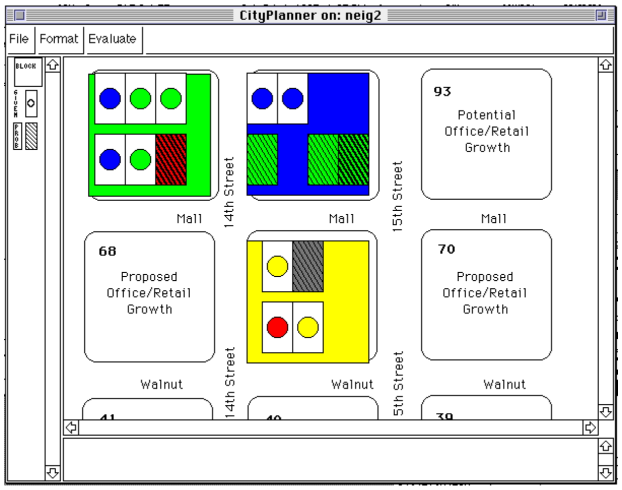


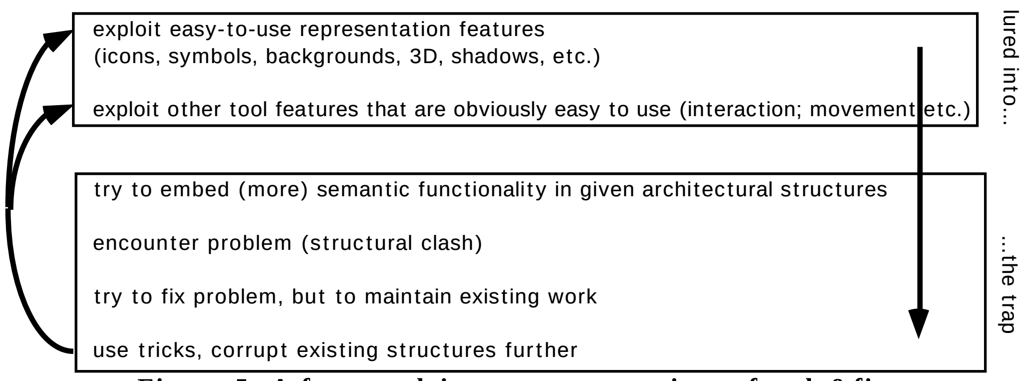
**Figure 4: vis-A-vis Application CityPlanner**

## The Deceived Designer

When a designer is deceived, many problems arise at the same time:

- Fixing a problem tends to mess up architectural structures; if the fix is again driven by easy-to-use features, the problem may get even worse (see SESAMsheet example).
- Designers and customers get used to rapid progress at the beginning. When features have been exploited and the more application-specific aspects of the system must be attacked, the perceived slow-down seems dramatic. In this situation, designers refuse to abandon parts of their work even if it is clearly premature. We learned from interviews (and, in some cases, introspection) that they feel this would slow down the process even further.
- Problem fixing further corrupts structures and makes progress even more difficult in the future.

Note that the broadcast problem (SESAMsheet) would be solved by an easy-to-use mechanism in vis-A-vis, and the adjacency problem of CityPlanner would not exist in Agentsheets. Exchanging the underlying application frameworks would avoid those particular problems, but probably encounter others.


exploit easy-to-use representation features
(icons, symbols, backgrounds, 3D, shadows, etc.)
exploit other tool features that are obviously easy to use (interaction; movement etc.)
lured into...
try to embed (more) semantic functionality in given architectural structures
encounter problem (structural clash)
try to fix problem, but to maintain existing work
use tricks, corrupt existing structures further
...the trap
**Figure 5: A feature-driven process; variant of code&fix**

The problems cannot be denied by blaming a particular framework. Complex applications *will* cause problems. The question involves when they will occur.

### Process Phases and Motivation

As illustrated by the examples and described in this section, a feature-driven process can unfold, as follows:

Only if the application-specific part is trivial enough can the lower three steps of Figure 5 be avoided. Framework designers probably do not see this danger as they tend to create examples and applications that are *well supported* by the framework. From their perspective, this seems a reasonable approach, since they want to demonstrate the features of their framework.

Qualitatively, designer motivation follows the curve in Figure 6: a long-term decline.

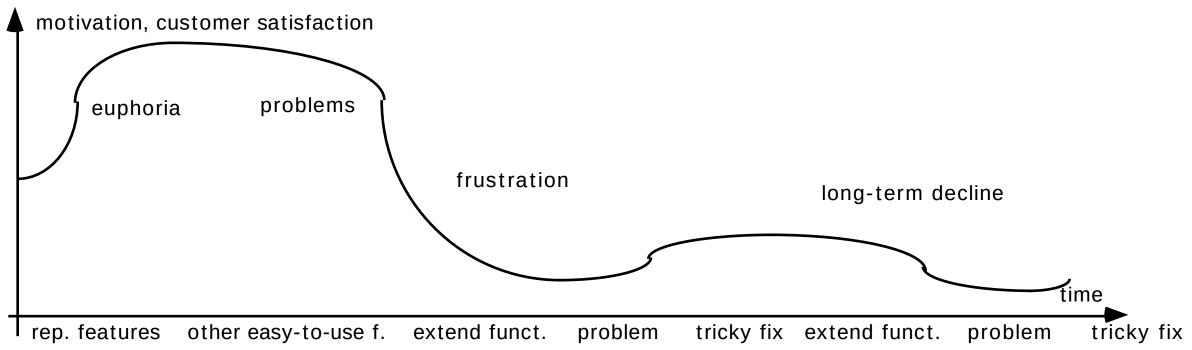


**Figure 6: Project progress and motivation decrease**

## 4 A Hidden Layer in Application Frameworks

There is a gap between a concrete application and the features and mechanisms of a given framework. A feature-driven process searches its way from the given low-level (detailed) features and mechanisms to the application. If the application fits perfectly into the framework, the gap is small. Most features of the application can be covered by straightforward combinations of features. In this case, no problems occur, and the feature-driven process is successful.

In more complicated applications, the gap is wider. Easy-to-use features provide a rapid start, but do not cover some of the crucial requirements and expected functionality of a concrete application. The combination of features and language constructs is nontrivial. Without proper technical design, the process can get stuck in a code&fix trap.

A first approximation to avoid the trap is *demanding* a disciplined design process. Such a process should deliberately start at the application and provide a full design of the concrete application before any code is written, and especially before any easy-to-use feature is exploited. Implementation (and design evolution, if necessary) should take a risk-driven approach (Boehm, 1988). Instead of implementing the easiest parts first, *the most critical and unclear issues must be solved first*. Convenient features can still be exploited later; it will not take long.

Demanding discipline is easy. However, it puts the burden of acting against instincts (sense of completion and closure) entirely on the application designer's shoulders. It is preferable to *encourage* a risk-driven approach. Framework designers can do this by providing a middle-level layer of reusable design building blocks (e.g., "design patterns": Gamma et al., 1994). Application designers can pick up those building blocks, and start constructing their application from larger components. Adapting details is easy: the goal of the adaptation is clear, and easy-to-use features help to work on details.

Although this sounds promising in principle, we found that application designers had a difficult time identifying complex reusable blocks in practice (Repenning, 1993). Reusable components usually are abstract in order to be reusable. Application designers simply could not map their applications to the abstract mechanisms supplied. The middle layer seems to be hidden and not easily accessible (Curtis et al., 1988).

Our suggestion is to provide access from the level of concrete applications to those abstract building blocks (Fig. 7). Access can be afforded by full applications that serve as examples and by metaphors that are common to a set of concrete applications. Working from analogous applications down to their adaptation proceeds from large overall structures to details. This fits much better into a risk-driven approach than blind code&fix ever can. We will go into more detail about paradigmatic applications and underlying metaphors in the next section.

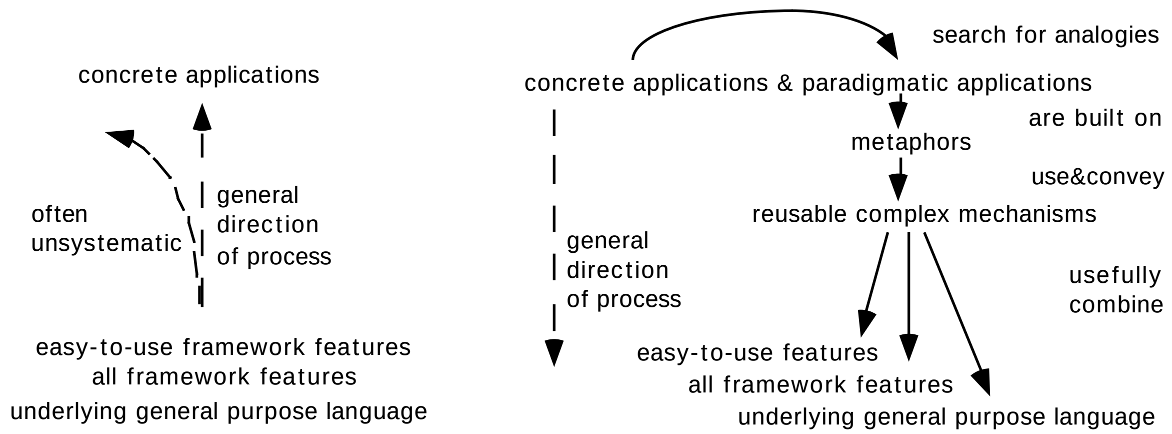

concrete applications
often unsystematic
general direction of process
easy-to-use framework features
all framework features
underlying general purpose language
search for analogies
concrete applications & paradigmatic applications
are built on
metaphors
use&convey
reusable complex mechanisms
general direction of process
usefully combine
easy-to-use features
all framework features
underlying general purpose language

**Figure 7: Feature-driven design process (left) and exploitation of a hidden middle-layer of reusable complex mechanisms to reverse process direction and to make it risk-driven (right)**

## 5 Paradigmatic Applications, Shallow Analogies and Metaphors to Identify Reusable Mechanisms and Design Building-Blocks

In the previous section, we identified a special purpose of examples: they must provide *cognitive access* to abstract building-blocks and reusable mechanisms. Many examples provided in manuals have other goals: they illustrate how features are *used*. For our purpose, we want to illustrate how features are combined and *used together* to obtain a higher-level mechanism. The difference may seem subtle, but it has significant practical consequences, as discussed in the previous section. Straightforward feature examples further promote features and push a feature-driven design approach. Abstract examples are hard to relate to a concrete application. Arbitrary applications, on the other hand, may not contain reusable mechanisms (Curtis et al., 1988).

In the remainder of this section we describe the kind of examples we consider most helpful to avoid deception by easy of use. We also outline how framework designers can find them and how application designers can access them. The key is to use metaphors and "shallow analogies" for both purposes.

We characterize the examples we have in mind as *paradigmatic applications*.

A ***paradigm*** is "1: example, pattern. 2: an example of a conjugation or declension showing a word in all its inflectional forms." (Webster). Paradigms, in their original meaning, are used in grammar books to convey an abstract grammatical pattern. See the paradigm of the Latin word "servus" (servant).

serv-us
serv-i
serv-o
serv-um
serv-e
serv-o

We claim that good examples for application frameworks are like paradigms:

1. They are *concrete, useful applications*, not abstract patterns for the sole purpose of illustrating features (serv-us, not only -us);
2. They are *lean* applications, showing few complications other than the one conveyed by the pattern (serv-us, not hippopotam-us).
3. They are *readily associated with the mechanism* they want to illustrate (servus always reminds the authors of this declension).

4. They show a clear distinction between the abstract pattern and the concrete application (the dash in serv-us);
5. They are mostly accompanied by an explanation of the pattern, but the paradigm alone is sufficient to remember the crucial points (the explanation would name the modes: nominative, genitive etc; would explain whether the two -o terminations have a common root). Usually, the explanation defines the scope of the paradigm.
6. Once you know a paradigm, it is easy to apply the pattern to other instances within the paradigm scope (hippopotamus is not really a problem).

It is not obvious that any application designer will associate a given application with exactly the pattern it wants to convey (3); this is a problem that also hits "design patterns" (Gamma et al., 1994). There are two ways to make the transfer easier:

- Explicitly: The pattern or mechanism can be explained, and the scope defined (5).
- Implicitly: As explanations and definitions of abstract patterns or mechanisms also tend to be abstract, it may help more to provide a number of analogous cases.

Analogous cases must not be detailed; there is no need to apply the paradigm to hippopotam-us, circ-us, etc. But they illustrate the scope. The dash helps to distinguish the pattern from the particular case.

Applying these notions to our domain of application frameworks, we propose:

- Framework designers should provide a number of paradigmatic applications.
- They should also provide a much bigger number of "shallow analogies": it is enough to mention another instantiation of the mechanism and identify the overlapping (analogous) parts. Implementation of analogies is not necessary.

Shallow analogies with explicit explanations of how they are analogous to the fully implemented paradigm are superior to a big number of fully implemented applications of the same paradigm. A bigger number and more implementation details make it more difficult to see the common mechanism and to recognize it as the same.

A ***metaphor*** is "a Figure of speech in which a word or phrase literally denoting one kind of object or idea is used in place of another to suggest a likeness or analogy between them (as in the ship plows the sea)" (Webster).

Formally speaking, a metaphor constitutes a mapping between two domains to stimulate associations. Common mechanisms that qualify for reuse can often be characterized by metaphors. When several applications are mapped to the same other domain by metaphorically speaking about them, this target domain is a good candidate for a paradigmatic application. Other domains that share the target domain of the metaphor should be provided as shallow analogies. This increases the chance to identify appropriate applications.

For example, the flow of water through pipes could be implemented as a paradigmatic application. In Agentsheets, it conveys the abstract mechanism of *propagating agents through a discrete space constrained by conductors*.

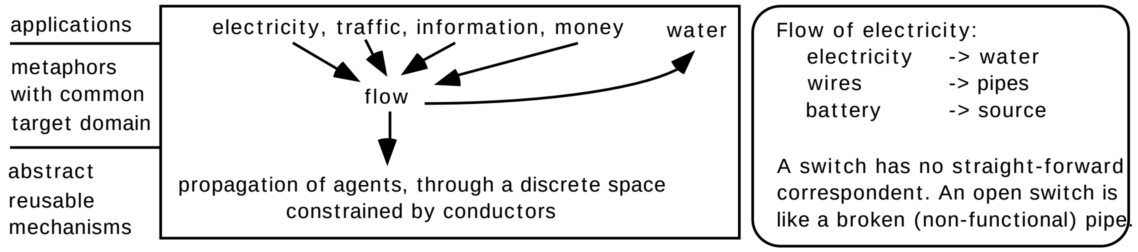

applications
metaphors with common target domain
abstract reusable mechanisms
electricity, traffic, information, money
water
flow
propagation of agents, through a discrete space constrained by conductors
Flow of electricity:
electricity -> water
wires -> pipes
battery -> source
A switch has no straight-forward correspondent. An open switch is like a broken (non-functional) pipe.

**Figure 8: Target domain of metaphor (water) qualifies as paradigmatic application (left); other domains with the same target domain should be provided as commented, shallow analogies (right)**

The analogies of traffic flow and electric flow should be explicitly explained, but not elaborated. A waterflow paradigmatic application could have supported both the electricity (flow of electricity, Fig. 1) and the SESAMsheet (flow of information, Fig. 2) application-building efforts.

The SESAMsheet problem could have been avoided or solved with an abstract *broadcast* mechanism, using a radio ("Public National Radio") paradigmatic application. In vis-A-vis, some application involving neighbors ("a chat with the neighbor" or "neighborhood watch") could convey the abstract notion and mechanism of *spatial adjacency*. Other examples help to identify what broadcast and spatial adjacency are good for. In any realistic setting, there should also be more domain-oriented analogies. A short explanation of the mapping is crucial; without it, some of the shallow analogies in Figures 8 and 9 may be ambiguous or unclear.

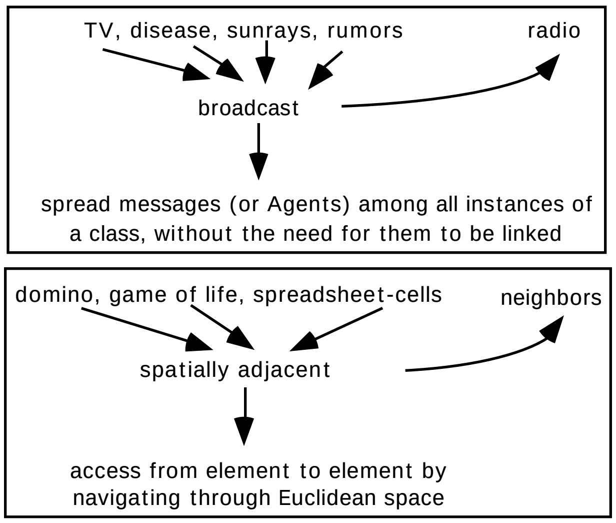

TV, disease, sunrays, rumors
radio
broadcast
spread messages (or Agents) among all instances of a class, without the need for them to be linked
domino, game of life, spreadsheet-cells
neighbors
spatially adjacent
access from element to element by navigating through Euclidean space

**Figure 9: Mechanisms and paradigmatic applications that would have solved our example problems**

Reusable mechanisms do not necessarily require new features. The broadcast mechanism could be implemented without any additional features. It is the knowledge of how to properly combine existing features in nontrivial ways that a reusable mechanism should convey. Reusing the alignmentgrid of vis-A-vis to define spatial adjacency requires intimate framework knowledge and the confidence that the grid is qualified for reuse. A framework designer is much more familiar with the appropriate and intended use of features than an application designer. Framework designers will themselves benefit from developing complete applications and analogies; this process can help them see the shortcomings and traps of their framework.

## 6 An Analogy-Driven Design Process

Boehm's (1988) advice to apply a risk-driven approach is still valid. However, it is especially hard to follow when easy-to-use mechanisms exist only on a low level, close to the framework. Just as it must have been fascinating to see computers doing *anything by just writing some lines of code* in the early days of programming, it is today fascinating to see a computer providing *any graphical interaction by just using some easy features*. Both temptations lead to unsystematic code&fix, a trap.

In a risk-driven design process, one would deliberately refuse to do anything that appears easy; just the contrary of what instincts dictate. Such a process relies heavily on designers' discipline, which is difficult to enforce or support.

We see an analogy-driven design process as the best approximation to risk-driven design:

- It makes application designers think about bigger structures first.
- It makes clear that something can be reused (maybe copied), but that it probably must be adapted (specialized, modified). This directs attention to high-level *structural problems* early.
- Reuse of higher-level building blocks implies adapting their low-level attributes only after they have been identified as generally useful. The exploitation of low-level features is thus deferred.
- The lack of a matching paradigm or shallow analogy may awaken the suspicion of application designers as to whether using the framework is really such a good idea.

All there are four points show that an analogy-driven design process tends to unfold from the application to the framework. An application designer may still have to combine several mechanisms, and invent new ones. But, his early attention has shifted to structural, risky issues.

A psychological advantage of an analogy-driven approach is that it shifts problems to the early phases of application design, and saves some motivational boosts for later. Identifying an analogy, seeing it at work, and finally customizing it with application-specific artwork spreads the sense of completion over the entire process, rather than wasting all the momentum at the beginning.

By all means, application designers should exercise discipline: not every application can fit into any given frame. It is wise to find the soaring points early: to decide about suitability and to discuss difficult problems as long as there is time. Paradigmatic applications, together with explicit analogies, can do much to facilitate (even partly afford) this process. Trivial (well-fitting) applications are hardly slowed down by an analogy-driven approach: it never hurts and it sometimes helps.

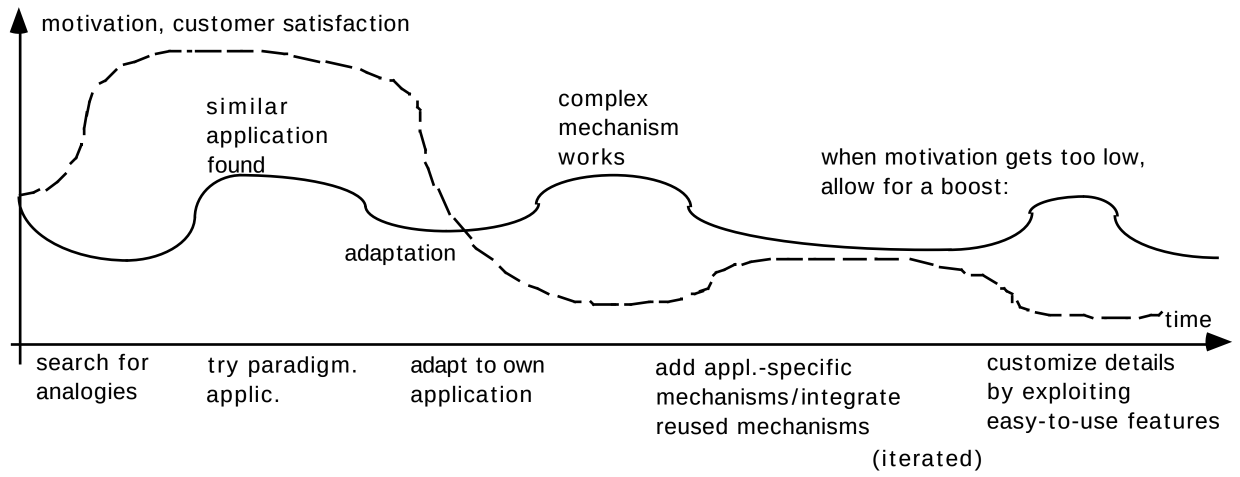


**Figure 10: Motivation in an analogy-driven process (solid line) shows no continuous decline and less extreme values than a feature-driven process (dashed line)**

## 7 Related Work

The value of examples is widely acknowledged. Fischer *et al.* (1992) suggest large collections of examples, which they call catalogs. We endorse their principle claim for an example-base. However, examples are not particularly helpful if they are too shallow, overcomplicated, or too similar. We claim that it is not as important to provide *many* examples as it is to provide *adequate* examples that address the hidden layer of visual design environments. An excessive number of more or less similar examples can even be counterproductive, as they aggravate the problem of identifying an adequate one. Adequate examples for application frameworks have not been characterized yet.

Nardi and Zarmer (1993) propose visual formalisms as an intermediate layer in user-interface design. By providing a small set of nontrivial data structures, designers can reuse both visual representations and some semantic operations defined on the abstract formalism. A tree is one example of a visual formalism. It implies not only a visualization, but also parsing, subtree selection, and some other operations meaningful for each tree. We fully agree with this concept, but want to generalize it and thereby extend its scope: In visual design environments, there are many design patterns common to a class of applications that cannot be reduced or mapped to interface representations. In a more specific domain (visual design environments instead of systems with a graphical interface), more specialized support can be given. Temporal and dynamic aspects, for example, are not at all captured by the static formalisms Nardi describes. Flow, our example in section 5, is not related to any static formalism. We also stress the importance of instantiating those abstract mechanisms within applications, and providing cognitive access from other possible applications by explicit analogies.

Gamma et al. (1994) present the concept of design patterns and provide a collection of those patterns. This collection is not meant to be "complete" in any sense; the development of domain-specific design patterns are encouraged. When provided in application frameworks, design patterns are located on a middle level of abstraction. They have very much in common with the abstract building blocks on the hidden layer of application frameworks that we describe in section 4. We argue that paradigmatic applications based on metaphors, and shallow analogies can make appropriate design patterns easier to identify for both framework designers and application designers. The description of a design pattern must show how it is instantiated in a concrete application Gamma et al. show such instantiations, but do not base the selection of their instantiations on metaphors; nor do they provide shallow analogies (except very brief pointers to other known instantiations of a given pattern). Our technique of factoring out common target domains of metaphors as domains for paradigmatic applications can also be applied by framework designers to identify what rewarding design patterns could be.

## 7 Conclusion

Framework designers tend to provide easy-to-use mechanisms on the basis of low-level framework features. Features of application frameworks for visual design environments are naturally concentrated around graphical representations and maybe interaction primitives. Application designers, on the other hand, can be deceived by low-level, feature-based features. Because graphical representations and interactions are the most prominent part of their applications, designers are easily lured into a feature-driven design process. Their sense of completion is tricked by the overwhelming visual prominence of graphical representations. Rapid, but unstructured, initial progress can turn into a trap if the application turns out to also have complicated *internal* structures. Rapid progress at the beginning of the design process must eventually be paid for by backtracking, stagnation of progress, corruption of application architecture, and frustration.

We introduce paradigmatic applications, together with a set of shallow analogies, as a way to reverse the design process direction. We want application designers to attack difficult problems first. To achieve a more risk-driven design process, both framework and application designers must contribute. We propose:

- Framework designers should encourage application designers to start their design process at a higher level. By examining paradigmatic applications first, common mechanisms on the "hidden layer of visual design environments" may be detected and reused.
- A large number of analogous applications should be provided, but not detailed. They help to identify the most suitable abstract mechanism.
- Despite all efforts from framework designers, application designers must avoid doing first what *looks* most impressive. Not even the most sophisticated set of paradigmatic applications and analogies can compensate for the fact that for each framework there are applications that do not fit the framework. Application designers must not rely on the illusion of guidance by framework features and follow an entirely opportunistic design process. It is often advantageous to deliberately follow a risk-driven design process.

We do not argue against evolutionary design or opportunistic planning *per se*. However, we are convinced that those concepts should not be used as a pretext only to indulge in easy-to-use features at once. A simple, linear process of requirements analysis and (off-line) design should be applied whenever it is sufficient – at least to get started. The slow and tedious process of evolution should be reserved to resolve difficult problems in which requirements are initially unclear (Floyd, 1984). Neither evolutionary design nor opportunistic planning must be abused as an excuse to neglect systematic solution of *technical* design problems resulting from the need to map an application to the application framework's construction paradigm.

## Acknowledgments

This work has been made possible by a research fellowship from the German Academic Exchange Service (DAAD), NSF (No. RED 925-3425 & Supplement), ARPA (No. CDA-940860), and Apple Computer Inc. We want to thank our colleagues at the University of Colorado at Boulder, and at the University of Stuttgart/Germany, who have used vis-A-vis and Agentsheets in numerous projects. vis-A-vis would not exist without Horst Lichter.

## References

Arias, E.; Anselin, L. (1982): A Modular Integrated Framework for Impact Assessment and Policy Analysis for Cities. In Vogt and Mickle (eds): Modeling and Simulation 14, Univ. of Pittsburgh

Boehm, Barry W. (1988): A Spiral Model for Software Development and Enhancement; IEEE Computer, vol. 21, pp. 61-72

Curtis, B.; Krasner, H.; Iscoe, N. (1988): A Field Study of the Software Design Process for Large Systems. *Communications of the ACM,* vol. 31, pp. 1268-1287

Fischer, G.; Girgensohn, A.; Nakakoji, K.; Redmiles, D. (1992): Supporting Software Designers with Integrated Domain-Oriented Design Environments; IEEE Transactions on Software Engineering, vol. 18, no. 6, June

Floyd, C. (1984): A Systematic Look at Prototyping; in: Budde, Kuhlenkamp, Matthiassen, and Züllighoven (eds.): Approaches to Prototyping. Proc. Working Conference on Prototyping, Springer, Berlin, pp. 1-18

Gamma, E.; Johnson, R.; Helm, R.; Vlissides, J. (1994): *Design Patterns,* Addison-Weseley, Reading, Mass.

Gamma, E.; Weinand, A.; Marty, R. (1989): Design and Implementation of Et++, a Seamless Object-Oriented Application Framework. Structured Programming, vol. 10, no. 2

Golin, E.J.; Danz, S.; Larison, S.; Miller-Karlow, D. (1992): Palette: An Extensible Visual Editor; Proc. of the 1992 ACM/SIGAPP Symposium on Applied Computing, pp. 1208-1216, March

Green, T. R. G. (1989): Cognitive Dimensions of Notations, Proceedings of the Fifth Conference of the British Computer Society, Nottingham, 1989, pp. 443-460

Lichter, H.; Schneider, K. (1993): vis-A-vis: An Object-Oriented Application Framework for Graphical Design Tools; Proc. of the IFIP Workshop on Interfaces in Industrial Systems for Production and Engineering, Darmstadt, Germany, March 15-17, Elsevier

Ludewig, J; Bassler, Th.; Deininger, M.; Schneider, K.; Schwille, J. (1992): SESAM - Simulating Software Projects; Proceedings of the Software Engineering and Knowledge Engineering (SEKE) Conference, Capri, Italy

McIntyre, D.W.; Glinert, E.P. (1992): Visual tools for generating iconic programming environments; Proc. of the IEEE 1992 Workshop on Visual Languages, VL'92, pp 162-168

McWhirter, J.D.; Nutt, G.J. (1994): Escalante: An Environment for the Rapid Construction of Visual Language Applications. Proc. of the 1994 IEEE Workshop On Visual Languages, VL'94, pp. 15-22

Nardi, B.A.; Zarmer, C.L. (1993): Beyond Models and Metaphors: Visual Formalisms in User Interface Design; Journal of Visual Languages and Computing, iss. 4, pp. 5-33

Norman, D. A. (1990): The Design of Everyday Things. Currency/Doubleday, New York

Repenning, A., & Sumner, T. (1995). Agentsheets: A Medium for Creating Domain-Oriented Visual Languages. IEEE Computer, vol. 28, iss. 3, pp. 17-25

Repenning, A. (1994). Bending Icons: Syntactic and Semantic Transformation of Icons. Proceedings of the 1994 IEEE Symposium on Visual Languages, St. Louis, MO: IEEE Computer., pp. 296-303

Shneiderman, B. (1989). Direct Manipulation: A Step Beyond Programming Languages. In R. M. Baecker & W. A. S. Buxton (Eds.), Human-Computer Interaction: A Multidisciplinary Approach, Morgan Kaufmann Publishers, INC. Los Altos, pp. 461-467