IPR No.: IPR2016-00500
Patent No. 7,864,163

EXHIBIT 1004

Reification, Polymorphism and Reuse: Three Principles for Designing Visual Interfaces

Michel Beaudouin-Lafon and Wendy E. Mackay
University of Aarhus, Department of Computer Science
Aabogade 34
DK-8200 Aarhus N - Denmark
+45 89 42 56 44 / +45 89 42 56 22
{mbl, mackay}@daimi.au.dk

ABSTRACT

This paper presents three design principles to support the development of large-scale applications and take advantage of recent research in new interaction techniques: *Reification* turns concepts into first class objects, *polymorphism* permits commands to be applied to objects of different types, and *reuse* makes both user input and system output accessible for later use. We show that the power of these principles lies in their combination. Reification creates new objects that can be acted upon by a small set of polymorphic commands, creating more opportunities for reuse. The result is a simpler yet more powerful interface.

To validate these principles, we describe their application in the redesign of a complex interface for editing and simulating Coloured Petri Nets. The cpn2000 interface integrates floating palettes, toolglasses and marking menus in a consistent manner with a new metaphor for managing the workspace. It challenges traditional ideas about user interfaces, getting rid of pull-down menus, scrollbars, and even selection, while providing the same or greater functionality. Preliminary tests with users show that they find the new system both easier to use and more efficient.

Keywords

Design principles, reification, polymorphism, reuse, direct manipulation, instrumental interaction, interaction model.

1. INTRODUCTION

Today's visual interfaces suffer from an overabundance of functionality: each successive version is marketed based on the number of new functions, with little regard to the corresponding increase in the cost of use. Simple things keep getting harder, as users spend more and more time deciding among an increasing variety of rarely or never-used options. Some users are at a breaking point and are less and less able to cope with new software releases [21]. Others have begun to actively reject software upgrades and cling to older versions of products such as Microsoft Word (survey of Microsoft users, Business Week, 5 July, 1999).

New interaction techniques, such as toolglasses [4] and marking menus [17], have been proposed to reduce this trade-off between power and ease-of-use. Yet such interaction techniques tend to be developed in isolation, as the focus of a particular research project. While this is a critical first step, it is also important to understand how these techniques scale when combined with other techniques and are placed in the context of complex real-world applications. We also need to develop new interaction models that explain how these and other techniques can increase the functionality available to users without creating a corresponding increase in the cost of use.

This paper describes how three design principles, reification, polymorphism and reuse, have provided a framework for redesigning a complex tool for editing and simulating Coloured Petri Nets. Developed in the late 1980's, the Design/CPN tool used a then state-of-the-art WIMP (windows, icons, menus, pointing) user interface. The new tool, cpn2000, is the result of a participatory design process, in which users and designers have collaborated to recreate a tool that supports "Petri-Nets-In-Use". The goal is to provide Coloured Petri Nets developers with greater functionality through an interface that is more intuitive, efficient and pleasant to use; one that allows them to think in terms of Petri nets and not the mechanics of the interface.

We begin by describing the principles of reification, polymorphism and reuse and then describe the interface to cpn2000. We explain how these principles have influenced the design of the user interface and discuss how combining them helps address the trade-off between power and ease-of-use. We conclude with directions for future research.

2. DESIGN PRINCIPLES

Graphical user interfaces can be broadly defined as consisting of graphical objects and commands. Graphical objects are represented on the screen and commands can be applied to create, edit and delete them. Visualization techniques describe how to represent these objects while interaction techniques describe how to apply commands to them. Over time, users develop individual patterns of use that depend upon the available objects and commands, the particular application domain and the current context of use. The perceived "ease-of-use" of an interface depends upon many factors, including the effectiveness of the visual representation, the completeness of the command set and the support for efficient patterns of use.

AVI 2000, Palermo, Italy.
© 2000 ACM 1-58113-252-2/00/0005..$5.00

We have developed three principles that address the issues surrounding objects, commands and patterns of use:

- *Reification* extends the notion of what constitutes an object;
- *Polymorphism* extends the power of commands with respect to these objects; and
- *Reuse* provides a way of capturing and reusing patterns of use.

2.1 Reification

Reification is the process by which concepts are turned into objects. For example, in a graphical editing tool, the concept of a circle is represented as an image of a circle in a tool palette. Reification creates new objects that can be manipulated by the user, thus increasing the set of objects of interest.

Instrumental Interaction [1] extends the principles of Direct Manipulation [26] by reifying commands into *interaction instruments*. An interaction instrument is a mediator between the user and objects of interest: the user acts on the instrument, which in turn acts on the objects. This reflects the fact that, in the physical world, our interaction with everyday objects is mediated by tools and instruments such as pens, hammers or handles. The menu items, tool buttons, manipulation handles and scrollbars seen in today's user interfaces are examples of interaction instruments. A scrollbar, for example, is both a visible object on the screen that can be manipulated by the user and also a command the user manipulates to scroll the document.

Turning commands into objects provides potentially infinite regression. Since instruments are objects, they can be operated upon by (meta)-instruments, which are themselves objects, etc. In real life, we see limited chains of regression, as we move our focus from pencils, to pencil sharpeners that sharpen pencils to screwdrivers that fix pencil sharpeners. In some user interfaces, menus and toolbar buttons can be reconfigured to tailor the interface: they become instrument objects that can be manipulated by meta-instruments.

Another example of reification is the notion of *style*: In a text editor such as Microsoft Word, a style is a collection of attributes describing the look of a text in a paragraph, e.g., the font and margins. The user can create and edit styles and apply them to paragraphs. Styles thus become objects of interest for the user.

Many graphical editors also reify a collection of objects into the notion of a *group*. Since a group is itself an object, it can be added to a group, giving way to arbitrarily large structures such as trees and DAGs. These structuring mechanisms can be found in a wide variety of interfaces.

2.2 Polymorphism

Polymorphism is the property that enables a single command to be applicable to objects of different types. Polymorphism allows us to maintain a small number of commands, even as reification increases the number of object types. This property is essential if we want to keep the interface simple while increasing its power.

Most interfaces include the polymorphic commands *cut*, *copy* and *delete*, which can be applied to a wide variety of object types, such as text, graphics, files or spreadsheet cells. *Undo* and *redo* can also be considered polymorphic to the extent that they can be applied to different commands.

Applying a command to a group of objects involves polymorphism at two levels. First, any command that can be applied to an object can also be applied to a group of objects of the same type by applying it to each object in the group. Second, any command can be applied to a heterogeneous group of objects, i.e. objects of different types, as long as the command has meaning for each of the individual object types.

2.3 Reuse

Reuse can involve previous input, previous output or both. Input reuse makes previously-provided user input available for reuse in the current context. For example, the *redo* command lets users repeat complex input strings without having to retype them. Output reuse makes the results of previous user commands available for reuse. For example, *duplicate* and *copy-paste* let users avoid re-creating complex objects they have just created.

Polymorphism facilitates input reuse because a sequence of actions can be applied in a wider range of contexts if it involves polymorphic commands. Prototyping environments such as Self and its Morphic user interface framework [22], which are based on cloning and delegation, support and even encourage a high level of input reuse.

Reification facilitates output reuse by creating more first-class objects in the interface which are then available for reuse. Thus, for example a Microsoft Word user can create a new style object by reifying the style of an existing paragraph or by duplicating an existing style object, modifying the copy and reapplying it. A more elaborate form of reuse obtains when new styles are created through inheritance from an existing style, which allows changes made in the reused object to be propagated to the edited copies.

Macros, such as those found in Microsoft Excel, illustrate the power of combining these three design principles. The user begins by telling the system to "watch" as a sequence of commands is performed. Reification enables the user to capture the particular pattern of use as a sequence of commands that can be applied as a single new command to a new set of objects. A more advanced form of reification turns each component command into an object that can itself be edited, thus changing the pattern of use to accommodate different contexts.

The next section briefly describes the cpn2000 interface, which provides a testbed for exploring these three principles.

3. THE CPN2000 INTERFACE

The current cpn2000 interface was created over a period of ten months by a group of ten people. We followed a highly participatory design process beginning with observation of users of an existing system, Design/CPN, in various work settings. We developed scenarios to capture and articulate their work practices and engaged in a variety of video brainstorming and video prototyping exercises to develop the new interface. These activities involved a multidisciplinary group of user interface researchers, programmers and Coloured Petri Nets developers. The first version of cpn2000 was presented at the CPN International Workshop in October 1999. We also took advantage of the CPN Workshop and an earlier retreat for the University of Aarhus CPN group to conduct more formal studies using CPN developers who were not involved in the development of the new tool.

The following sections introduce the basic concepts and vocabulary of Coloured Petri Nets (CPN), the basic interaction techniques we selected and the overall design of the interface.

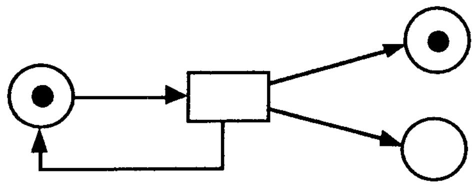

Fig. 1: A simple Petri net with three places, one transition and four arcs. Two places have a token.

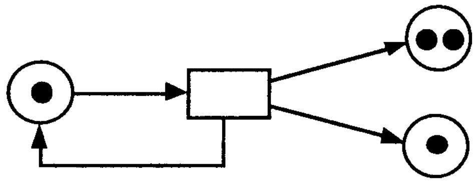

Fig. 2: The Petri net from Fig. 1 after the transition has been fired.

3.1 Coloured Petri Nets

Both Design/CPN and its successor, cpn2000, address the application domain of editing and simulating Coloured Petri Nets [14]. Petri nets are a graphical formalism with a strong underlying mathematical model that extends the power of simple finite state automata. Petri nets are particularly suited for the modeling and analysis of parallel systems such as communication protocols and resource allocation systems.

The graphical representation of Petri nets (Fig. 1) is a bipartite graph where the nodes are called *places* (depicted as circles or ellipses) and *transitions* (depicted as rectangles). Edges of the graph are called *arcs* and can only connect places to transitions and transitions to places. Each place typically represents a possible state or resource of the system. Places hold tokens, which represent the fact that the system is in a given state or the number of resources that can be allocated. The rules for simulating the net are very simple: a transition is *enabled* if all the places connected to it by an input arc have a token. *Firing* an enabled transition consists of removing a token from each input place and adding a token to each output place of the transition (Fig. 2). Mathematically, a Petri net can be represented by a matrix and simulation of the net is equivalent to a set of linear algebra operations. Properties of the net can be proven, such as the fact that the net has a bounded number of tokens or that there are no deadlocks.

A number of higher-level Petri net formalisms have been developed to model complex systems. Most of these formalisms are equivalent in power to a simple Petri net, but are much more concise. One such extension is Coloured Petri Nets [14]. In this model, the tokens belong to a *color set* equivalent to a data type in a conventional programming language. Arcs are labeled with pattern-matching expressions that describe which tokens are used when a transition is fired. Typically, colors allow a conventional Petri net to be "folded" onto itself, making models much smaller. In addition Coloured Petri Nets can be hierarchical. A transition can be described by a subnet, equivalent to macro-substitution in a textual language. Hierarchical nets make it possible to structure a complex net into smaller units that can be developed and tested separately.

Over the past decade, the CPN group at the University of Aarhus has been developing an editor and simulator for Coloured Petri Nets, called Design/CPN (Fig. 3). This tool is freely available to the CPN community and is currently in use by over 600

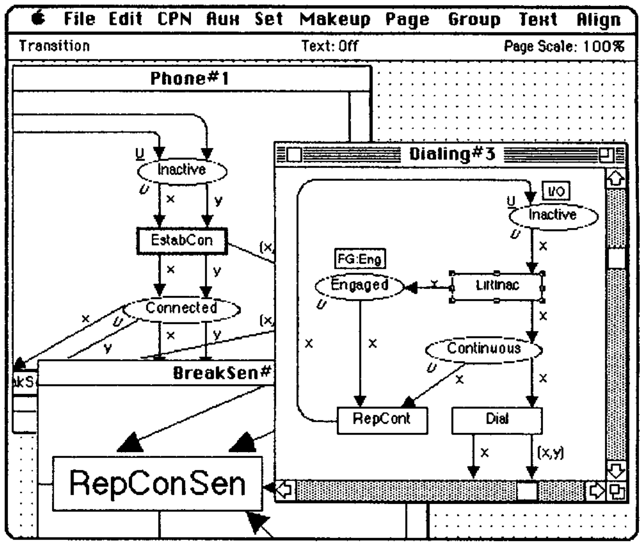

Fig. 3: Design/CPN, the current tool used by CPN designers.

organizations both in industry and academia. Design/CPN users have created models with as many as 100 modules and have run simulations lasting several days. The tool has been used far beyond the expectations of the designers and has reached its limits in terms of usability and complexity of implementation. The goal of cpn2000 is to reimplement the basic functionality of Design/CPN while improving the user interface and adding new editing and simulation capabilities. The project is a joint effort of the CPN, HCI and Beta groups at the University of Aarhus and is funded by the Danish Center for IT Research, Hewlett-Packard and Microsoft.

3.2 Interaction techniques

We began with two key decisions that have influenced many aspects of the design. First, we decided to explicitly support two-handed input, with a mouse for the dominant hand and a trackball for the non-dominant hand. The keyboard is used only to input text and to navigate within and across text objects. The design of the bi-manual interaction follows Guiard's Kinematic Chain theory [10] in which the non-dominant hand manipulates the context (container objects such as windows and toolglasses) while the dominant hand manipulates objects within that context. The exception is direct interaction for zooming and resizing, which, according to Casalta et al. [6], should give both hands symmetrical roles.

Second, we decided to incorporate a combination of new interaction techniques, rather than using a standard WIMP interface. Our goal is to provide cpn2000 users with easier yet more powerful tools and support more effective patterns of use. Users should be able to spend more time on developing Petri nets and less time on the mechanics of the interface.

The current version of cpn2000 incorporates four primary interaction techniques: direct interaction, marking menus [17], floating palettes, and toolglasses [4].

Direct interaction involves pointing directly at objects and either clicking on or dragging them. A direct bi-manual interaction, used for resizing and zooming, involves depressing a trackball button with the non-dominant hand and dragging the mouse with the dominant hand, as if stretching a piece of rubber.

Marking menus are radial, contextual menus that appear when clicking the right button of the mouse. Marking menus offer faster selection than traditional linear menus for two reasons. First, it is easier for the human hand to move the cursor in a given direction than to reach a target at a given distance. Second, the menu does not appear when the selection gesture is executed quickly, which supports a smooth transition between novice and expert use. Kurtenbach and Buxton [17] have shown that selection times can be more than three times faster than with traditional menus. Hierarchical marking menus involve more complex gestures but are still much more efficient than their linear counterparts.

Floating palettes contain tools represented by buttons. Clicking a tool with the mouse activates this tool, i.e. the user conceptually holds the tool in his or her hand. Clicking on an object with the tool in hand applies the tool to that object. In many current interfaces, after a tool is used (especially a creation tool), the system automatically activates a "select" tool. This supports a frequent pattern of use in which the user wants to move or resize an object immediately after it has been created but causes problems when the user wants to create additional objects of the same type. cpn2000 avoids this automatic changing of the current tool by getting rid of the notion of selection (see below) while ensuring that the user can always move an object, even when a tool is active, with a long click (200ms) of the mouse. This mimics the situation in which one continues holding a physical pen while moving an object out of the way in order to write.

Toolglasses, like floating palettes, contain a set of tools represented by buttons. Unlike floating palettes, they are semi-transparent and are moved with the non-dominant hand. A tool is applied to an object with a *click-through* action: The tool is positioned over the object of interest and the user clicks through the tool onto the object. The toolglass disappears when the tool requires a drag interaction, e.g., when creating an arc. This prevents the toolglass from getting in the way and makes it easier to pan the document with the non-dominant hand when the target position is not visible. This is a case where the two hands operate simultaneously but independently.

Since floating palettes and toolglasses both contain tools, it is possible to turn a floating palette into a toolglass and vice versa, using the right button of the trackball. Clicking this button when a toolglass is active drops it, turning it into a floating palette. Clicking this same button on a floating palette picks it up, turning it into a toolglass.

None of the above interaction techniques requires the concept of selection. All are contextual, i.e. the object of interest is specified as part of the interaction. This greatly simplifies the application's conceptual model and, one hopes, the users' mental models. However, this also creates a problem. Traditional interfaces use multiple selection to apply a command to a set of objects. We solve the problem by reifying multiple selection into objects called groups (see below).

We considered several other interaction techniques including gesture input [25], zoomable interfaces [2] and dropable tools [3]. We selected the above set partly due to the participatory nature of our design process, which led us to select the techniques most appealing and natural for our particular set of users. However, the techniques we chose also cover each of the different possible syntaxes for specifying commands:

- *object-then-command:* point at the object of interest, then select the command from a contextual marking menu;
- *command-then-object:* select a command by clicking a tool in a floating palette, then apply the tool to one or more objects of interest;
- *command-and-object:* select the command and the object simultaneously by clicking through a toolglass or moving it directly.

Preliminary results from our user studies [13] make it clear that none of these techniques is always better or worse. Rather, each emphasizes a different, but common, pattern of use. Marking menus work well when applying multiple commands to a single object. Floating palettes work well when applying the same command to different objects. Toolglasses work well when the work is driven by the structure of the application objects, such as working around a cycle in a Petri net.

3.3 Workspace manager

Coloured Petri Nets frequently contain a large number of modules. In the existing Design/CPN tool, each module is presented in a separate window and users spend time switching among them. Early in the project, it became clear that we had to design our own window manager to improve this situation: the Workspace Manager.

The workspace occupies the whole screen (Fig. 4) and contains window-like objects called *folders*. Folders contain *pages*, each equivalent to a window in a traditional environment. Each page has a tab similar to those found in tabbed dialogs. Clicking the tab brings that page to the front of the folder. A page can be dragged to a different folder with either hand by dragging its tab. Dragging a page to the background creates a new folder for it. Dragging the last page out of a folder removes the folder from the screen. Folders reduce the number of windows on the screen and the time spent organizing them. Folders also help users organize their work by grouping related pages together and reducing the time spent looking for hidden windows.

Cpn2000 also supports multiple views, allowing several pages to contain a representation of the same data. For example, the upper-left page in Fig. 4 shows a module with simulation information, while the upper-right page shows the same module without simulation information but at a larger scale.

The left part of the workspace is called the *index* and contains a hierarchical list of objects that can be dragged into the workspace with either hand. Objects in the index include toolglasses, floating palettes and Petri net modules. Dragging an entry out of the index creates a view on its contents, i.e. a toolglass, a floating palette or a page holding a CPN module.

Pages and folders do not have scrollbars. If the contents of a page is larger that its size, it can be panned with the left button of the trackball, even while the dominant hand is using the mouse to, for example, move an object or invoke a command from a marking menu. Getting rid of scrollbars saves valuable space but makes it harder to tell which part of the document is currently visible. A future version will display relative position information on the borders of the page during the panning operation in a non-intrusive and space-saving way.

Resizing a folder and zooming the contents of a page involves direct bi-manual interaction (as described above). Unlike traditional window management techniques, using two hands makes it possible to simultaneously resize and move a folder, or pan and zoom the contents of a page at the same time. Clicking the mouse on the page tab or on the folder pops up a contextual marking menu with additional commands, such as close, duplicate, collapse and expand.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | sales@docketalarm.com | 1-866-77-FASTCASE

fastcase®
Smarter legal research.