Unifying Software Elements with LISP-based Object-Oriented Technology

Markus Fischer, Symbolics Systemhaus GmbHGermany[1]

Abstract: The paper describes the use of object-oriented techniques throughout all layers of software, in the design and the implementation phase. These techniques increase the productivity of the software engineer as well as the user's benefit from the application. Several tools and their interdependencies are described and supported with some examples. The resulting methodology is needed in order to match today's needs of innovative industrial users.

Keywords: Common Lisp, Object-Oriented Technology, Software Engineering, ODBMS, CLOS, CLIM, Statice

1 Introduction

Many problems in industrial areas are still unsupported by computer applications, although it would often be very profitable. One reason for this can be found in the complexity of the problems to be solved. Conventional approaches revealed themselves to be insufficient in most of such cases, because of their general limitations; more innovative techniques have to be used instead. The object-oriented paradigm allows us to develop software that goes beyond that borders of conventional paradigms.

2 Motivation

People see objects around them all the time, and they handle and manipulate these objects.
This way of thinking is adapted in the object-oriented approach. That is, the OO paradigm is cognitively adequate.
Consequently, user interfaces that consider this human attribute are more often accepted than conventional ones. The desktop simulation of the Apple Macintosh or the innovative NeXT-Step interface are just two examples of the tendency to make the work with the computer more intuitive and easier. Still, both examples just deal with a restricted set of modelled "real world" objects. For example, the Apple Macintosh has a well-designed interface for handling files and programs - the so called Finder, but the programs that run on the Macintosh might not be so intuitive, e.g. a planning system.
That is, new problems necessitate new classes of objects, featuring new behavior.

As stated above, the user has high expectations about tbe computer system that should ease his working load.

---

[1]The author can be reached via E-Mail: MF@SGER.uucp

Likewise, a software engineer has expectations about the process of software development: The program should be easily and quickly designed, coded, and maintained. A part of his work is to model the world of objects of the user, i.e., to map these objects to data and code. Thus, object oriented tools can be quite a supportive way of doing this, in contrast to conventional techniques. Especially in the Lisp market, there are many powerful object-oriented tools to choose from.

In order to develop and use software successfully, it is not enough to have a technologically appealing concept. The present economical situation, which is rather difficult world-wide, most companies are forced to cut costs heavily, in order to stay profitable or reduce losses. This cost reduction also applies to software development. No matter how fancy a system might be - the bottom-line is the financial advantage resulting from using it. Therefore, the software has to be built quickly, yet taylor-made, and easily exensible for future demands of the user. It has been proven in many published cases that the object-oriented paradigm can be successfully employed for this purpose in prinicipal. However, we emphasize in our work, that the usage of this technique in the overall process of software engineering and all parts of a program, actually multiplies this advantage.

3 Conceptual Elements of Software

There are several parts of software belonging to conceptually different categories; among them are:

Long Term Data	Usually called a *database*, which in most cases includes features as persistency, concurrent access from multiple workstations, and the need of consistency even after a mashine crash. In some cases normal text files - indexed or sequential - are used. Those files are slowly substituted by databases due to their advantages.
Short Term Data	These are data typically created and deleted in a high frequency, locally used, and only of temporal relevance, thus stored in virtual memory - not explicitly on disk.
Procedures	Some kind of operations that can be executed in order to manipulate the long or short term data.
User Interface	A visual, often graphical interface for interaction with the user.

3.1 Data

Though in principal short and long term data can be divided into two different categories, these concepts have many things in common. They are structured and organized in a

hierarchy. Encapsulation, information hiding, and a clearly defined access interface are important for both kinds of data. But in most systems presently developed, there are two distinct formalisms and two ways of dealing with short term data and long term data. To avoid this overhead, short and long term objects, and the classes they belong to, should be described and manipulated the object-oriented way. Organizing the database different to the objects in virtual memory necessitates an unnecessary second way of thinking for the application programmer, and furthermore, he possibly won't be able to model the user's world of objects adequately anymore. There are also problems occuring on the code level, for example, where data has to be transferred, respectively converted from one kind of storage to the other. In our projects with industrial partners, this is a knock-out criterion for a database product - because the cost simply rises for software production when using many types of storage systems. This is true for the process of production, maintainance, and documentation. Another important factor making object-oriented storage systems attractive is performance: The more complex the class framework and the structure of the objects get, the less performant get non-object-oriented databases, e.g. relational ones. This results from the semantic difference between the object-oriented and the relational model. There are several studies and reports revealing this partly enourmous performance advantage. In the publications [3] and [4] actual benchmark results for representative database applications can be found.

As an example for the integrative view to database and virtual memory objects, consider a class *flight*, with its objects stored in a database, in order to be accessed by several flight planners simultanously - with just one at a time having the right to write information about a flight. Concurrency (control) is needed here, as well as persistency. Some other data, like objects that serve as collections of currently selected flights must be just data in virual memory, because they are only locally used and just relevant for a short time. But from the application programmer's point of view, the data should be accessed and manipulated the same way.
A major role in the realization of this methodolgy is played by CLOS, the Common Lisp Object System, which is part of the emerging ANSI standard for Common Lisp. A detailed description of CLOS can be found in [6].

Objects stored in CLOS can only be used as language for short term data, because those objects live in virtual memory. Furthermore, there is no predefined support of indexed access, concurrency, or data consistency and recovery in CLOS. To have this functionality, Statice, an object-oriented DBMS can be used. Statice is tightly integrated in Common Lisp, e.g. through the retrieval language and type system. It is actually almost completely written in Lisp; and, what is very important for our integrative approach of object-oriented software design and implementation, Statice uses the same way as CLOS to describe, access, and manipulate classes and objects. Features like encapsulation, accessor functions, multiple inheritance, and many others are also part of Statice. The concept of generic functions and methods is actually taken from CLOS. That is, one can invoke CLOS generic functions to conventional CLOS objects that live in virtual memory, as well as to Statice objects that are actually stored on a database server's disk. The only thing needed to get this behavior, is to specialize methods to the appropriate class of objects, either a Common Lisp type, a CLOS class, or a Statice class.

3.2 Procedures

In Lisp, code is partitioned into functions. The concept of generic functions introduces the possibility to define a single interface for a function, still having the ability to specifaclly define the behavior of every class of objects being passed as arguments. For example, it might be feasible to have a class *flight* with two subclasses: *domestic flight* and *international flight*, where international flights allow a departure or arrival airport to be from world wide set of airports, whereas for domestic flights both airports have to be in the same continent or country. A generic function that checks the vailidity of airports for a certain flight, then might have two participating methods defined for the classes named above. Adding, redefining, or removing classes of objects does not cause any problems in that approach, because the interface for the function stays the same. Code is compact and can be maintained easiliy - even by people that were not involved writing the software. In addition to that, a program can be set up from libraries holding a set of classes & methods. These libraries can be easily set up and extended by extracting code from an existing program, because the object-oriented approach automatically supports the right way of modulization.
Using Common Lisp (including CLOS) plus the additional facilities of Statice is the adequate way of realizing the desired technique of a homogenious code in Lisp.

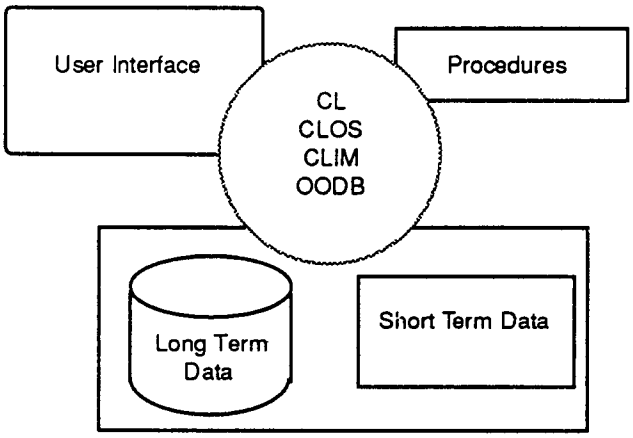

Figure 1: CL-related products for transparent object-orientation

3.3 User Interface

The user thinks in the terms of objects that in the whole make up his working place. The software engineer maps these real world objects into data that might be stored either in a database or in virtual memory as described in the previous sections. The user interface is the basis of interactions between user and system. One task of the user interface is to visualize objects in a way the user can easily recognize its semantics. Visualization is one issue, but cognitive user interaction also includes adequate handling of user gestures. While formerly designed systems just covered keyboard input, newer designs

like the one of the Macintosh also deal with mouse actions. This is a very helpful trend in the computer industry - even programs runing on DOS-PCs slowly turn to this interaction style. Still, in some cases the connection between application objects and their visualization and operations might not be tight enough. What problems exist and what we are doing to solve them is described in the following two sections.

3.3.1 From the User's Perspective

A sophisticated direct-manipulative user interface usually incorporates icons, which represent objects the user wants to manipulate; buttons (in some contexts called gadgets or widgets); and commands, which are normally displayed in pull-down or pop-up menus. Still, for the user it is not totally intuitive a) which objects he can manipulate and b) what operations are possible with those objects.

Even though buttons or gadgets can look quite fancy, they normally are not part of the user's world of objects, that is, they are artificial objects which need explanation. Consequently, the user can not use the software written for his working area, without considerably changing the mental model of his work. In some cases, this leads to total rejection of the system or even worse, to reduction of motivation and efficiency - which is exactly the opposite of what had to be achieved with the system.
Furthermore, even in advanced user interfaces currently used, the number of directly manipulative objects is relatively small, and in most interactions the user has to deal with conventional patterns of *<Operation> <Arguments...>*.

The user needs a set of virtual objects on the screen that are as similar as possible to the ones he already knows. The same is true for the operations that are invoked on these objects. This was taken into account in the design of CLIM, the Common Lisp Interface Manager. For example, objects are not only considered as operands for operations; mouse clicks on them are interpreted as operations - what operation is invoked by mouse click depends on the applications current situation. This interaction style is totally conform to the user's perception and thinking.
For people that prefer typing commands instead of using the mouse - mainly very experienced users do so, a fancy, state-of-the-art command processor can be included as another interaction style; even mixed on the level of operations and operands - without changing a single line of code.

3.3.2 From the Software Engineer's Perspective

Even when using direct-manipulative user interfaces, there is usually an additional layer between the application data and the visualization which cannot be bypassed. The advantages using the object-oriented paradigm for application data and procedures are almost totally lost at the user interface level, because in conventional systems, the UIMS is using a non-object-oriented paradigm for the interaction as well as for the actual

implementation and its programming interface. In the worst case, the programmer has to deal with bits, bytes, and pixels - instead of objects, as he does in its application code which is not directly involved with user interaction.

However, there are tool boxes available or under development that try to reduce these kinds of problems by giving the programmer the possibiliy of placing buttons, sliders and other related UI objects on a window. In general this is a good idea. Still, these tools don't really visualize application objects - they usually introduce their own model of objects.

On the other side, there is a long tradition of object-oriented user interface design and implementation in the Lisp environment. There it is taken into account that there is a strong and very useful tendency of making a system more and more transparent and understandable to the user, giving him/her more and better ways to interact with it.
Lately, CLIM - the Common Lisp Interface Manager was established in the market as the de-facto standard in the Lisp community[2]. It turned out to be highly powerful and adequate for the employment in projects for industrial customers. It uses exactly the same object model as CLOS in order to have a single formalism for transparent and consistent application and user interface development. See the section "Implementation" for details about integration of CLIM with CLOS and the other described object-oriented tools.

4 Analysis and Design

Programmers and users normally "negotiate" about the functionality of an application during the phases of analysis and design. The programmer initially has difficulties to think the way the user does about his work. On the other side, the user often does not know what can be achieved by employing computer technology. Prototyping that uses the object world of the user, has proven to considerably reduce time and efforts in that phase of the application development. See [1] and [2] for a more detailed description of the advantages of using the object-oriented paradigm in prototyping.

For the purpose that are subsumed in the concept CASE, there are some tools available on the market that try to support the use of concepts like classes, objects, attributes and relationships as recommended from experienced people in the OO communitiy, e.g. Ed Yourdan. Employing these tools, we observed some disadvantages and deficiencies, and so we decided to build tools ourselves for some special aspects of CASE that incorporate also the implementation-related aspects of an object-oriented design. Although many things still have to be done, we find it an interesting way to go. One of such tools acts as workbench for classes, attributes, and relationships that can be manipulated graphically using an object-oriented user interface. It supports abstract views to a schema[3] as well as language-specific restrictions or extensions - not necessarily those of Lisp. The designed schema can be transformed in executable code just by "pressing a button", in order to have short time intervals between consecutive steps of evolutional software development.

---

2 See [5] or [7] for an introduction to CLIM
3 A *schema* is a coherent collection of classes optionally connected by relationships.

5 Implementation

5.1 Used Tools

The embedding language is normally Common Lisp. It provides the level of abstraction that is needed to keep up with object-oriented concepts. Furthermore, it is easily extensible and supports a dynamic behavior of software. In our software developments we use the object-oriented constructs introduced by CLOS.
It is one of the very few state-of-the-art OO-systems among the many so-called object-oriented languages that just include selected parts of object-oriented paradigm, namely those that fit the static structure of these languages. In contrast to that, CLOS is quite powerful and expressive, yet highly portable and efficiently implemented. Symbolics Systemhaus uses CLOS in almost every project for industrial customers on varying platforms.
The fully object-oriented DBMS Statice uses the same concepts of CLOS and provides the optimal functionality and structure of a database system in the framework of CL/CLOS. It is currently available for Symbolics platforms and will be available on Unix machines at the end of 1992.
On the User Interface side there is CLIM. It is like CLOS available for a wide set of platforms. Similar to Statice, it incorporates the constructs and concepts of CLOS.
It also provides all of the best ideas of Dynamic Windows, the Symbolics-proprietary UIMS, including the Presentation Type Model.

5.2 Examples

In the following, a few short examples for some excerpts of the described methodology are given to further support and explain it.

Object - Visualization - Operation

Due to the fact, that objects can be *presented*, which roughly means to bring the application object on the screen, the user directly manipulates the data of program; no additional layer is introduced. As an example, an object of class *flight* can be visualized by a couple of graphical elements and a short textual identifier, e.g. the flight number. The user easily recognizes it as a flight, and the user, e.g. a flight planner intuitively perceives the object as known. That is, there is a directly experienced match of the user interface and the user's mental model of his usual working area.

Invoking an operation on an object can be done by several ways using the mouse: The operation
is mapped on one of the mouse buttons which can be invoked when locating the pointer over the object. The operation can be choosen from a menu of operations that is popped up by clicking on the object, or by clicking first on the visual representation of the operation and then on the objects that act as operands.

CLIM already provides mechanisms for all that cases - the programmer just declares, what operation (*command* is the appropriate CLIM term) is available for what classes of objects.
For an operation, that, say, shows how late a flight is relative to its scheduled time, it does not matter, what the flight looks like or by what gesture the operation was invoked. The view that supports object-oriented thinking and programming is that *there is* an operation or method invoked on an application object - meaning that there is just one piece of code for the function's semantics; independent from its invocation or representation of it or its operands.

CLOS objects - Statice objects

Using CLIM, there is no difference between CLOS objects and Statice objects - both kinds are treated exactly the same way. For example, one is able to specialize so called *presentation methods* for CLOS classes or Statice classes. The term *presenting* roughly means to visualize objects on the screen.
The technique of using the same mechanism for objects in virtual memory and objects that reside on a database server's disk has two major advantages: 1) The code is very compact and generic. The programmer does not care about an objects source - database or virtual memory. 2) The user can operate directly on objects that are not in virtual memory, maybe not even on his own workstation, but actually on the disk of a remote server withouth introducing a second way of thinking, perception, or manipulation of objects.
Like the UI methods of CLIM, e.g. the already explained presentation methods, CLOS methods can handle both CLOS objects and Statice objects - the code is designed, written, maintained, and documented just once.

Operations - Methods

Operations, i.e. CLIM commands are collected in command tables. These tables can be organized in a hierarchy with inheritance of commands - just like CLOS methods for classes. That similarity allows to have methods on the direct interaction level. In addition to the concept of visual, manipulative objects, this is the second major part of having a single concept of software elements from the database up to the user interface.
Example: A class *aircraft* may have two subclasses: *modifiable-aircraft* and *static-aircraft*, where there are some methods available for all objects of type *aircraft*, including both ot its subclasses: E.g. number-of-seats, maintance-time, etc. There might be also a method for changing the version of an aircraft, which means rougly to change the ratio of seats of business class and economy class. But this method is only allowed for modifiable-aircrafts. In our model, all the methods are also CLIM commands in command tables - one command table for each class with the identical inheritance as for the classes. The result is that the operation for changing the version is a command only available for displayed objects of class modifiable-aircraft. We call this concept *User Interface Methods*.

6 Discussion and Outlook

The proposed methodology of unifying software elements has proven to be a powerful and cost-effective way to deliver innovative software for industrial users with complex problems.
That way, Symbolics Systemhaus has designed and implemented a flight schedule planning system worth about $6 million for the German airline Deutsche Lufthansa AG, and other projects are underway. Also inhouse developments, like the mentioned schema designing tool, employ the described methodology.

The only possibility to implement these techniques nowadays, is to use the already standardized languages and tools Common Lisp, CLOS, and CLIM. The OODBMS Statice is also a major part of this strategy. But to make software not only unified object-oriented, but also 100% portable, which is an important factor in the development of software, some things are still left to do. Statice is not yet available for non-Symbolics platforms. However, the Unix version of it is to be expected early 1993. This will allow us to design and implement software using Genera, Symbolics' state-of-the-art Development Environment, still having the possibility to easily deliver on other platforms that are not as suitable for development as Genera, but can be used as cost-effective runtime-platforms.

At the user interface level, CLIM 1.0 has still some deficiencies in employing the look & feel of the host's window system, e.g. the one on the Macintosh, Open Look or Motif on Sun and other Unix machines. The soon available version 2.0 that is developed by all Lisp vendors in a close cooperation will finally eliminate that problem.

Compared to other systems and methodologies used in the industry today, the described way of object-oriented software engineering is the most expressive, transparent, and flexible we know about. The toolset CL, CLOS, CLIM, and Statice revealed themselves as the optimal software basis to implement applications using this approach, due to the powerful object-oriented features.
Both the methodoly and the toolset represent the keys for the success of the projects we designed implemented for our commercial and industrial partners.

Literature

[1] M. Grund, C. Popp: *Reguliertes Prototyping.* Internal technical report, Symbolics Systemhaus GmbH, Germany, 1991.

[2] M. Grund, G. Timmermann: *Supporting Complex Flight Scheduling Tasks Using Statice and CLOS.* To be published, Symbolics Systemhaus GmbH, Germany, 1992.

[3] Karin Erni et. al .: *Performance of Object Oriented Databases.* Available directly from ABB Heidelberg, Germany, 1992.

[4] R.G.G. Catell & J. Skeen: *Object Operations Benchmark.* ACM Transactions on Database Systems, Volume 17 No. 1, 1992.

[5] R. Rao, W.M. York: *A Guided Tour to the Common Lisp Interface Manager.* Lisp Pointers Volume IV, 1991.

[6] S. Keene: *Object-oriented Programming in Common LISP*, Addison-Wesley, 1989.

[7] Scott McKay: *The Common Lisp Interface Manager.* Communications of the ACM, Volume 34, Number 9, September 1991.