At Harlequin created on September 29, 1994 and last modified at Harlequin on November 8, 1994.

Overview of the DylanWorks Environment and Methodology

Roman Budzianowski & Scott McKay

1. Introduction

In this paper, we present an overview of the architecture of the DylanWorks environment and a high level vision of the methodology for developing software products using Dylan. This paper does not offer any specific solutions (those will be addressed in the other white papers), except for illustrational purposes.

2. Overall Goals of DylanWorks

The overall goal of DylanWorks is to create an integrated software development environment that allows a team (or teams) of people to create and manage a software product through all the phases of its lifetime. The environment will encourage the development of software that is based on reusable components. It will also support the process of "growing" software from an evolving set of requirements and prototypes into a final product. The final delivered product will not be encumbered by any residues of the development environment.

Of course, the process of developing and delivering software is cyclic. There is a starting phase where a set of goals and a product description and specification are developed. The long middle phase is taken up by writing and debugging code and documentation, which may result in revisiting some (or many) of the decisions made in the first phase. Finally, the product is tested (which testing has hopefully been going on during the middle phase as well) and shipped to customers. Customer feedback will almost always result in reexamining the goals and specification of the product. In any event, the next release of the product will require that the cycle be repeated. One of the goals of DylanWorks is to model the cyclical nature of this process so that information is not lost between phases or cycles of product development.

A long-term goal of DylanWorks is to provide a "total capture of everything" model, in which we use an object-oriented database to capture (either by directly including the information, or keeping references to information provided in other repositories, such as Frame documents) all requirements, design specs, design rationale, source code, bug reports, bug fixes, test suites, management entities (such as to-do lists), and so forth. The intent of capturing all this information is so that, over the lifetime of a project, people can answer such complex questions as "why was this decision made?".

An important point is that DylanWorks, as an environment, is not just a piece of software, but an implementation of a methodology. This methodology needs to be carefully thought out and care-

fully documented. Note that we are not right now proposing to provide direct support any particular object-oriented methodology (such as Booch).

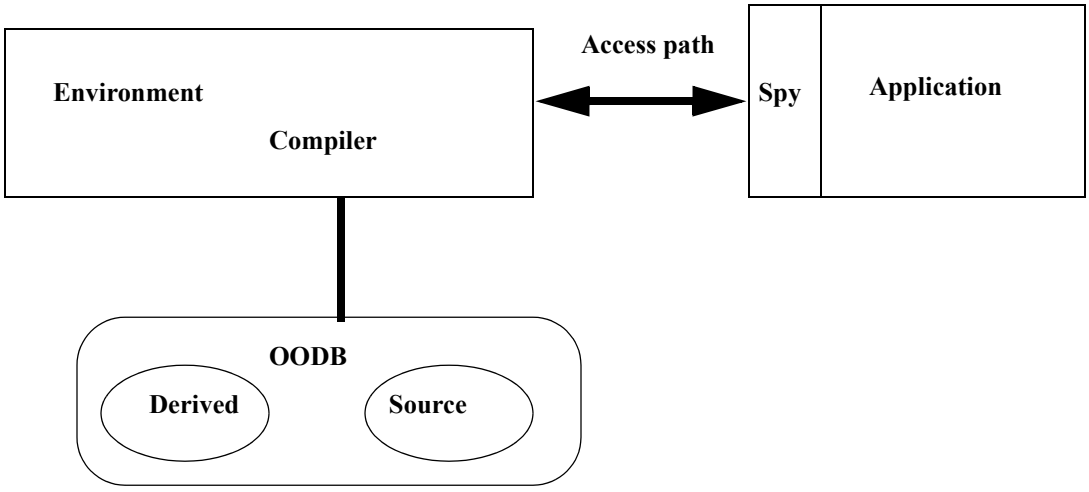
3. Overview of the Architecture

The main distinguishing characteristic of the Dylan environment relative to Lisp is that the program being debugged is generally run as a separate process, possibly on another machine. (We might have an option of running it in the same address space as the environment, but this is simply an optimization for the sake of convenience.) In this approach the environment is just another Dylan application. This model is closer to the traditional Unix modular approach than the monolithic Lisp environments. At the same time we intend to have the advantages of an interpretive environment. This would be achieved by using incremental compilation to implement a read-eval-print loop, and to provide fix-and-continue functionality in the debugger. Those features will be available in the development environment, and possibly in the runtime as a separate add-on libraries.

Another element of the Dylan environment is the persistent object database for storing the source code and information derived by the compiler and other tools. The source code database will provide a more convenient means for storing, versioning, browsing, and incrementally compiling the code then the traditional flat-file format, since it will operate at the granularity of a single definition. It will allow the user to annotate the code for various purposes: documentation, compiler hints, and so forth. At the same time, we will provide means for externalizing the source code in the flat file format for integration with the traditional environments. Accordingly, the compiler will be capable of running stand-alone and of processing the flat file format.

Figure 1, “Dylan Environment Architecture,” on page 2 shows the architecture of the Dylan environment.

FIGURE 1. Dylan Environment Architecture



Scott - please finish the picture.

4. Methodology

4.1 The Motivation

As the systems being built are becoming more complex we need a better way to visualize and organize the programs. Even Kent feels blind when working with LW, because he is used to Genera. Most developers would feel blind both in LW and Genera. This is especially acute when trying to learn a large system written by somebody else. Meta dot just doesn't do it. We need more graphical ways to present the program and to browse through it. Most of you feel very comfortable in a traditional Lisp environment, but remember that we are aiming our product at the market where Visual C++, Visual Basic and Power Builder dominate. I think that even experienced developers would benefit from better ways to view and organize programs.

Example. When at Oberon I had to learn a system of about 1 mln lines of code written in a home grown dialect of C++ with CLOS look&feel (methods were defined outside of class definition). The development environment, which included a preprocessor and build utility, supported the notions of a module and a subsystem. The preprocessor understood some dependency annotations and automatically genereted the h files. It was extremely difficult to learn the system. One reason was that I couldn't just look at a class and see all the interfaces (like in a traditional C++). The other was an extremely complex dependency structure. We had a graphical tool which was capable of displaying the subsystem dependencies but only in a static and non-interactive way. The result was similar to the class browser in LW. Too complex to be useful. And it didn't allow to look at the system at different semantic zoom factors.

4.2 Traditional Approach

Even though the notion of a flat file is not a first class concept in Dylan we are going to provide a file interface. On the lowest level, the file interface will be provided by the compiler accepting input from files. In this way a traditional environment can be used.

People voiced the opinion that there is a need for grouping things together and that files serve that purpose, i.e. they are used by the programmer to organize the program. In some sense, they are orthogonal to the module system (even though one cannot have files spanning modules). Since our environment will use a database to store source code we will provide a way to organize programs into files. In this way the text based development will be still possible. The current design of the source database (white paper) supports this.

4.3 Better Organization of Code

Smalltalk which is not file based provides two additional (outside of the language) concepts to help organize programs: category and protocol. The browser has four columns: (1)category is a set of classes, (2)class, (3) protocol is a set of methods, (4) methods. This classification is rigid, but still helpful.

I think that we should provide a flexible scheme for grouping things together. We might provide a standard classification, which would include files. The scheme should allow for both non-overlapping groupings and overlapping groupings. The user should be able to create new categories and name instances of them. I think that the source database should support such a flexible scheme instead of being based on the file/directory concept.

Examples of useful categories might be:

ClassGroup[overlapping], Interface[non-overlapping]

Then we can have ClassGroup[abstract] and ClassGroup[lib=streams]. Of course we already have libraries and modules in Dylan which provide some organization facilities and would interact with the user defined categories. In effect I am thinking of a “programmable” (but not in Dylan?) meta structure of the program.

This approach would overlap with the many OO analysis tools (like Booch) but could be potentially more felexible and powerful. Note that those tools, in effect, provide only a notational framework.

Note that I don’t advocate any extensions to the language. I sympathise with opinions that a lot of functionality can be provided in the environment.

4.4 Browsing and Editing Paradigm

The idea comes from Nicolas and Watson even though their (Lisp based) environment is different. If we store source in a database then a natural paradigm for browsing is DATABASE QUERY. As in Watson you could build a query visually. The query system could be expanded to allow for constraints.

Note that the organization system described above would work with the query system (in Watson only files are part of the database). This approach could be expanded to include not only browsing but developemnt and editing. In other words instead of opening a file to add a class or a method, one would manipulate a visual representation of a query in effect performing an update of the database. The query would be always kept up to date. The programmer may end up in the editor, but the important is the way he got there - not by guessing the right place. This aproach would also allow for a flexible grouping of code fragments depending on context and the user’s choice. For example:

interface[=read-from-stream]->gf->method->interface[=*]

This query would result in a graphical display of the dependences. The user may edit any part of it (e.g. a method) by selecting it and invoking the editor. The user may select more then one part and the editor displays the code together. Note that there is no rigid grouping in the editor. Selecting interface would display all methods in the interface.

Note that to perform such a query we need the derived information from the compiler. The compiler doesn’t know about “interfaces”, so we would have to do some additional work to include user defined categories in the query.

In Watson, the approach is used for analysis: a query is performed once and displayed like in a traditional database system. In Dylan I imagine that the user would construct several queries which would reflect the organization of the program he is working on. Those queries would always reflect the current state of the database without user intervention. I.e. if the user adds a new class,

all queries which would involve this class would be updated. This requirement might pose quite a challange in terms of performance.

4.5 Visualization

Watson is an example where visualization works for analizing Lisp programs. We would need to do more work in this area. The paper on 3-d visualization at the OOPSLA conference was very interesting. It attempted to show large scale pictures of systems. The base of the approach was an algorithm to draw 3-d pictures of objects. This was done by automatically generating shapes and colors from the names of the classes. If similar names generate similar shapes I can imagine that the user might find the visualization helpful in browsing large systems and analysing various dependences.

The basic idea is that we want to show a “big picture” of the system, which would show various dependencies. We also want to be able to zoom semantically to show more detail. One element of the Self environment I would use here is the “core dumper”, even though they use it in slightly different context. When you look at a large scale picture of the system you may see an overall organization and dependencies, but there is no detail. So you use a “magnifying glass”. What I liked about core dumper is that it was simpler and more convenient at the same time. There is a tool, which you pick up with the mouse and point at some element of the picture. You click and in another window (an inspector window) you see what is under the mouse in more detail. You can edit the thing right then and there. I think that the same technique could be used for debugging.

Note 1: as I mentioned there are OO analysis tools on the market whose only role, I believe, is to manage complexity by graphically displaying the system under construction and by providing a common notational framework. They also generate some C++ code. Our environment is persistent, not file based, so the notion of generating the code doesn’t really apply here. The user just edits the system.

Note 2: I saw Sillicon Graphic’s development environment for C++. It is very well done. In the debugger they have a 3-d display of arrays of numbers. It has a very limited use, but it was very neat to see how you can detect a problem. If the runtime structure of the program gets complex, it would be very useful to apply visualizations to runtime objects, both on the stack and in the heap.

4.6 Summary

I am suggesting that we use database query as a paradigm for looking at and browsing through programs. The query is constructed graphically. The result of the query is also presented graphically in 3-d using shapes, color and connections. The user may edit the program through this graphical representation either by direct manipulation (e.g. changing the class hierarchy) or by using the editor. The query is always kept up to date and represents a view of the program.

I am hoping that through this flexible query mechanism, the programmers would come up with their own custom organization/view of their program which would help them handle complexity.

Even though it requires some research and design, we have a base from which to start: Watson.

At Harlequin created on September 29, 1994 and last modified at Harlequin on November 8, 1994.