At Harlequin created on September 23, 1994 and last modified at Harlequin on September 5, 1995.

DylanWorks Design Papers

Scott McKay & Chris Fry

1. Introduction

The overall long-term 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 resuable 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.

The purpose of this paper is to outline a suite of short white papers that describe the DylanWorks environment. Each of these white papers is intended to address one portion of DylanWorks. Each white paper will:

- Identify its scope,
- Identify the overall strategy to be used in its area,
- Identify candidate solution(s), and
- Identify problems for which we have no good solutions yet.

2. Overview Papers

This set of white papers describes the overall requirements, goals, and design principles for DylanWorks.

Paper 1. Overall Requirements of DylanWorks — requirements.doc

This paper describes what the Dylan team thinks the overall requirements for DylanWorks is. There will presumably be many assumptions and questions that need to be answered by people in product management and marketing. For example:

- What hardware and OS will the typical user have?
- How much disk and memory will the typical user have?
- What other devices will he have? For example, what size display?
- What tools will he have used in the past?
- What kind of applications will he be writing?

Paper 2. Overview of Environment and Methodology — methodology.doc

This paper presents an overall map of the Dylan environment, whose "parts" include an application being worked on which is separated from the development environment itself. These two major pieces are linked via an "access path". This paper also describes the relationship between the compiler and the rest of the environment, how we will model a project via databases, and the interaction between the various tools.

Paper 3. Overall Class Library Design Guidelines — library-design.doc

This paper is basically a summary of available literature that describes how to do robust, extensible class library design. It also describes more prosaic things, such as the naming conventions used by the Dylan libraries.

Paper 4. Overall UI Design Guidelines — ui-design.doc

This paper describes overall guidelines and principles for the DylanWorks user interface, that is, its "look and feel". Some typical guidelines are:

- All the tools should fit together seamlessly and serendipitously.
- There should be uniform undo/redo everywhere.
- There should be context-sensitive help and documentation everywhere.
- All functionality should be accessible via both mouse and keyboard. It should scale to support other input devices, such as voice input.
- Hardcopy should be supported everywhere.

3. Project Databases

Libraries and applications built in DylanWorks are organized into "projects". Each project has its source code, documentation, specifications, and other additional information stored in databases.

Paper 5. Persistent Storage — persistent-store.doc

This paper describes the strategy that DylanWorks itself will use for the persistent storage of sucvh things as source code and derived information. Note that this is not necessarily the same thing that a customer will use as an object-oriented database; customer needs are more likely to be addressed by an interface to a commercial object database, such as ObjectStore.

Paper 6. Source Database — source-db.doc

This paper describes the source database. Initially, the source database will contain versioned source code with "simple" annotations (such as author, change comments, and so forth). In the long term, this will contain many other annotations, which are listed beelow. The initial implementation will probably be a "quick and easy" version control substrate; later, this will either be extended or we will use a third-party system such as ClearCase.

The source database will contain versioned source “sections” that are organized into “files” (i.e., ordered sets of sections). Each section tracks such things as its author and creation time, why changes were made, it’s status (completed, experimental, broken), and so on. Other tools can access the source database at the granularity of a single section. There are numerous kinds of sections, all organized in a web. Kinds of sections include, but are not limited to:

- Requirements and specifications
- Documentation
- Test suites and examples
- Bug tracking information
- Schedules and to-do lists

Of course, the source database will allow importing flat files or files from other version control systems (such as RCS), and will support exporting source into an interchange format, such as flat files.

Paper 7. Derived Information Database — derived-db.doc

This paper describes the derived information database, which comprises all of the information that can be derived by running a compiler (or some compiler-like tool) over the source code of a particular project. It contains such information as:

- Mapping from “definition names” to section names
- Source location (“meta-point”) information
- Mapping from program counters to source code fragments, and vice-versa
- Argument and local variable names and types
- Meta-object information for all types and classes
- “Who calls” and “calls who” information
- Macro-expanders and in-line function information
- Compiler warnings

4. Core Environment

The “core environment” comprises those tools that are required by essentially all users of DylanWorks to do their day-to-day work.

Paper 8. Project and Library Manager — project-manager.doc

This paper describes the “application” that controls the overall aspects of a DylanWorks session. The project and library manager allows a user to select what project (or projects) is being worked on, set the attributes of the current session (such as the compiler settings), and select an access path. In a sense, this tool provvides the entry into the rest of the DylanWorks environment.

The "librarian" aspect of this tool allows the user to organize a set of source into a library, and provides for some system configuration management. (In the future, we may choose to use an off-the-shelf product to do this, such as Atria's ClearCase.) Some of the the functionality includes:

- Management of libraries, which may be nested.
- "Journalling", that is, tracking the exact state of a project at any point in time.
- Integration with the source version control component of DylanWorks.
- Assistsing users in locating relevant functionality; this might be assisted by some sort of "agent" that determines what is in various libraries.
- "Costing" of functionality, so that users can see how expensive any piece of functionality is (in space and speed).

Paper 9. Compiler Overview — compiler.doc

This paper describes the overall functionality and organization of the compiler.

Paper 10. Edit/Compile/Debug Cycle — edit-compiler-debug.doc

This paper describes the DylanWorks edit/compile/debug cycle. The basic model is built on an incremental compilation/dynamic linking model; of course, certain "tightly compiled" applications may not support full incrementality, but the overall goal is to provide this facility during most of the development process. This paper also describes the concept of patching and the tools that support patching; such patching tools provides one aspect of groupware.

Paper 11. Editor — editor.doc

This paper describes various strategies for doing editing within DylanWorks. The strategies include using a third-party editor (such as Gnu Emacs or PFE), and writing some sort of editor from scratch. In a perfect world, the editor would include support for:

- "Hard" sections.
- Buffers that are constructed by explicitly linking together sections; in particular, buffers will have a finer granularity than a whole file.
- Each section, and each "line" in a section knows how to display itself; this allows for embedded graphics, gadgets and icons, "fish-eye" views, and so forth.
- Language and context-sensitivity, allowing for such things as templates, intelligent help, and so on.
- It will support multi-media, in that things like sound and video can be encapsulated in containers (such as OpenDoc parts) within a buffer.
- Other Emacs-like functionality, such as macros and an extension language.

Paper 12. Debugger — debugger.doc

This paper describes the both the substrate and some proposed functionality for the DylanWorks debugger, and the requirements the debugger puts on the compiler. This debugger will support a uniform model for debugging programs written in various languages. There will be the usual functionality, such as:

- A backtrace (stack) browser
- By-name access to argument and local values and types
- Evaluation of language forms in the appropriate context
- Breakpoints and watchpoints, and trap-on-exit
- Source-level stepper
- “Fix and continue” and restarting of frames

Paper 13. Listener — listener.doc

This paper describes the listeners, which are essentially a kind of “null” debugger that lets the user evaluate language forms. In particular, this paper discusses how interactions with the listener relate to the edit/compile/debug cycle and to modules and libraries (i.e., how definitions defined a listener can be captured in a module, or how debugging forms might be captured into a test case).

This paper also discusses a model for FFI-like remote evaluation and printing of “proxy” objects that underlies the listener.

Paper 14. Browsers — browsers.doc

This paper presents an overall “theory of browsing”, some proposed architectures for browser implementations, and describes a suite of browsing tools provided by DylanWorks. It also describes how the various browsers interact with each other in a tightly integrated way.

The DylanWorks browsers are intended to be extensible, but also provide some built-in “tools”. The built-in functionality provides easy access to:

- Dynamic data browsing (aka, inspector)
- Class browsing
- Generic function and method browsing
- Caller/callee browsing
- Project/library/module/source browsing, including “apropos”
- Thread and process browsing

5. Auxiliary Tools

The auxiliary tools in DylanWorks are those tools that, while still important, are likely to be used less frequently than the core tools.

Paper 15. Interface Builder — ui-builder.doc

This paper describes our strategy for providing an interface builder. We anticipate that this will, for the most part, be a fairly traditional GUI builder (at least in the initial release of DylanWorks), so there is a buy *vs.* build decision here. Note that, whether we buy or build, the GUI builder must be prepared to create OLE/OpenDoc compliant interfaces.

Paper 16. Profiling Tools — profiling.doc

This paper describes the DylanWorks profiling tools. We plan to break these tools into three phases:

- Data collection (including time and memory usage).
- Profiling data browser, which uses information hiding and filtering to allow a user to sift through the large amounts of data generated during a profiling run. The profiling interface will be built on the same substrate as is used by the other browsers.
- Protocol for feeding information back to compiler, and for feeding back cost information to the librarian.

Paper 17. Mail Reader — mail-reader.doc

This paper describes a strategy for integrating mail reading into the rest of DylanWorks. This will include some sort of “Chat” or “Converse” facility. Note that the initial release of DylanWorks in unlikely to include this.

Paper 18. Bug Reporting and Tracking — bug-tracking.doc

This paper describes our strategy for doing bug reporting and bug tracking. Note that the initial release of DylanWorks in unlikely to include this.

Paper 19. Application Packaging and Delivery — packaging.doc

This paper describes the phase of application development that “shrink wraps” an application for delivery. It addresses issues such as DLL creation, installers, logical pathname installation and other site management issues, and how delivered applications might be patched or otherwise modified.

6. Libraries

These white papers are *not* full specifications of the libraries, but are simply high-level descriptions of what the libraries will do.

Paper 20. Standard I/O Libraries — standard-io.doc

This paper describes the goals of a set of standard I/O libraries, which include:

- Streams, which includes support for streams over sequences and files
- “Stdio”, which includes support for Lisp-style ***standard-input*** and ***standard-output***

- Locators (aka, pathnames), which include support for URL pathnames
- Format

Paper 21. Threads — threads.doc

This paper describes the goals and issues of a multi-thread library for Dylan, including locking and fluid binding. This library will use native facilities wherever possible.

Paper 22. Memory Management — memory.doc

This paper describes the high-level goals of memory management and garbage collection in DylanWorks.

Paper 23. FFI and Interface Importation — ffi.doc

This paper describes the goals of the Dylan foreign-function interface library. The FFI part of the paper mainly describe the Dylan object representation, issues of data marshalling and unmarshalling, and function calling conventions. The Interface Importation part of the paper describes a facility for mostly-automated importation of foreign interfaces, such as those described in C include files.

Paper 24. User Interface Substrate — ui-substrate.doc

This paper describes the components of a high-level portable user interface library. This library will consist of several distinct layers, each of which can be used by any user application. From bottom to top, these layers are:

- A low-level, fairly platform-dependent layer that links directly to the native UI toolkits.
- A mid-level, portable layer that describes the traditional composition toolkit model of building user interfaces (i.e., events and gadgets, simple graphics, color, etc.); this will be based on the ideas of CAPI, Silica, and Fresco.
- A high-level, portable layer that allows programmers to describe their user interface in terms of the semantics of the application; this will be built on the ideas of CLIM.

The user interface libraries will also integrate with embedding protocols (such as OpenDoc or Ole) and scripting protocols (such as OSA or Ole Automation).

Paper 25. Environment Query (“Gestalt”) — gestalt.doc

This paper describes a library that provides a portable way to query the environment, for example:

- Host name and host type
- User name
- Current date and time
- Operating system type and version number
- Hardware configuration

This paper also describes a simple set of “compile-time” queries that can be used as “compile-time conditionals” in code.

Paper 26. OODB Interface — oodb.doc

This paper describes the strategy to be used for providing user access to persistence and object-oriented databases (such as ObjectStore).

7. Standards Integration

DylanWorks needs to integrate well with a number of emerging standards. Unfortunately, since some of these standards are still competing with one another in the marketplace, it is not clear which of them we should use. These papers outline the issues surrounding these standards, and what we might do.

Paper 27. CORBA Strategy — corba.doc

This paper discusses what options are available to us in the domain of common object representations. Options include DSOM (which is CORBA-compliant) and COM (Microsoft’s crack at objects, which is not yet CORBA-compliant).

Paper 28. Embedded Documents — embedded-docs.doc

This paper discusses what options are available to us in the domain of document embedding. It addresses the issues of both creating user applications that can act as “parts” or “part handlers”, and how DylanWorks might itself be composed of parts and part handlers. The primary, competing standards here are OpenDoc and OLE.

Paper 29. Internationalization Strategy — internationalization.doc

This paper discusses possible strategies for making DylanWorks internationalizable. It addresses such issues as Unicode.

Paper 30. Third-party Integration — third-party.doc

This paper discusses how we might integrate with 3rd-party tools and standards. This discussion is related to CORBA and OpenDoc/OLE issues. Such 3rd-party tools and standards include:

- Mail reading tools
- Version control tools (RCS, ClearCase)
- Configuration management tools (ClearCase)
- On-line documentation (Web, Frame, Acrobat)

8. Documentation

Paper 31. Documentation Issues — documentation.doc

This paper discuss how we will develop documentation, and how it will be accessed in DylanWorks. There are two separate issues here:

- Contents of the documentation, including scenarios and tutorials, a users guide, and a reference manual, all of which should be available both in hardcopy form and on-line (preferably through a hypertext book reader).
- Integration with environment, such as integration with the context-sensitive input facilities provided by the user interface substrate and support for “smart apropos”.

This will also discuss how DylanWorks users will create documentation for themselves and for their end-users.

9. Futures

Paper 32. Futures — futures.doc

This paper will describe some things that we are explicitly not dealing with in the first release of DylanWorks, but which we are considering for future releases. Such things include either producing or integrating with:

- Multi-media libraries
- Documentation editing tools
- Graphical editing Tools
- Project scheduling and management tools
- Bug tracking and QA tools

Other things we will not be addressing in the first release of DylanWorks include:

- Support for multiple programming languages
- Cross-compilation and cross-debugging
- Visual programming
- Design rationale capture
- CASE
- Generation of real-time and embedded applications
- True RPC and IPC
- ISO 9000