At Harlequin created on September 23, 1994 and last modified at Harlequin on November 11, 1994.

DylanWorks Requirements

Roman Budzianowski, Chris Fry & Scott McKay

1. Introduction

This paper outlines the basic goals we hope to achieve with DylanWorks. It describes a set of models that describe our target market, users and organizations, the types of applications they might develop, and what sort of hardware they are likely to use. For each of these models, we describe some requirements dictated by the model. We then go on to describe a "feature space", that is, a set of technologies we might implement to address each possible target.

Finally, we describe a plausible single product we can build that addresses a likely set of targets. More detailed discussion of all the technologies found in that product are described in the other DylanWorks white papers.

Note that this paper is far from the ultimate market analysis and requirements document. However, it does serve to provide the DylanWorks development team with a starting point and a basis for making technical decisions. It is our intention that management and marketing people will read this document so that they can understand why we are making certain decisions; we hope that these people will also do some market research to either back up these decisions, or help change them if need be.

Note that, even more than the other white papers, this is an evolving document reflecting our model of the current state of affairs.

2. Vision for DylanWorks

DylanWorks consists of a set of tools that assist developers in writing software systems, starting by building a small prototype, cycling through further requirements analysis and prototypes, and finally resulting in a delivered application. The "final" application itself will be maintained using these same tools. The application might be a self-contained free-standing Dylan application, or it might be a modular component (either a library or an entire application) that integrates well with other tools and languages.

We envision that DylanWorks will aid the entire development and maintenance cycle, in part by providing "hypertext" links between all aspects of a project, eventually including such things as design rationale capture. The initial release will support source version control and system configuration management, both batch and incremental compilation, and patching. Later releases of DylanWorks will model specifications and documents as well as code, will track bug reports and test suites, and so on.

The intent is to make the transition from developing the prototype(s) to delivering the finished product is intended to be as smooth as possible. In DylanWorks, there will be no huge effort at the

end of the development process required to deliver a product. Furthermore, ongoing maintenance will not be an “afterthought” process supported by out-of-band tools, but will instead be an integrally support part of DylanWorks.

Note that experience in the Lisp community should serve as a warning that it is not enough for our development environment to be very good only at assisting developers in producing excellent “prototypes”, without being equally good at finishing the job. It needs to be the case Dylan as a language is a viable solution for delivering shrink-wrapped applications. In addition, these applications need to interoperate very well with existing software (such libraries written in C, OpenDoc/OLE, *etc*.).

2.1 DylanWorks Goals

Put another way, DylanWorks has to be very good at three things:

- Be powerful enough to make both individual programmers and teams of programmers very productive.
- Deliver small, fast, reliable shrink-wrapped components.
- Deliver applications that integrate very well with existing software and standards.

In addition, DylanWorks needs to be fairly good at this:

- Integrates well with the platform it is running on.

3. First Release of Product - executive summary of choices

This is not carved in stone, but here is a vague definition of the first release. Vague - because we presented here some alternative directions. Various options are discussed in “Models” section on page 6 and “Feature Space” section on page 7.

3.1 Overall direction

We have to decide what is going to be the selling point of DylanWorks (maybe we can find a better name). Due to resource and time limitations we probably have to prioritize our development strategy. The following areas are potentially important for the success of the product.

- Development UI (browsing, editing, debugging)

  The conservative approach would be to develop a traditional Lisp-style environment. It is powerfull but probably inaccessible to our target audience. It is not novel by any means and probably looses, even if it’s very good, with the likes of Visual C++.

  More aggressive approach is to develop a graphical, interactive environment which would help manage the complexity of programs. This would attract attention to our environment and lower the learning curve for novice programmers.

- UI Runtime Support

  Support for developing GUIs in Dylan.

  Conservative approach is to interface with a GUI builder on Windows and provide a library to hook up the code to the GUI.

  More ambitious approach would require better integration of the GUI builder with the development environment, probably necessitating development of our own GUI builder. The integration would involve a novel approach for interactive, direct manipulation, maybe prototype-based building of User Interfaces.

  Another ambitious project in this area is to provide a total UI portability solution.

- Application Embedding and Distribution Standards

  The conservative approach is to provide good interface with OLE 2.

  The ambitious approach is to attempt to interface with OpenDoc, D/SOM, CORBA. Note that those standards are still in flux.

- Interoperability

  The ambitious approach is to develop a full FFI to C and C++, with subclassing between C++ and Dylan both ways. Less ambitious approach would allow for limitations in the interface to C++.

- Tools

  GUI builder is part of the UI runtime solution. Other potentially useful tools are: schema editor and form generator for relational databases, various libraries.

We probably cannot take the ambitious approach in each area. We suggest that the development UI is very important in view of the success of Visual Basic, etc. We have to provide a standard support for runtime UI, but developing our own GUI builder which is significantly better than the existing ones is a big project. A complete portability solution is a big project too. We suggest that these projects be included in the next releases of DylanWorks. Another area that is very important for the success in the target market is interoperability: we need a complete FFI to C++. This has a very high priority and requires a lot of resources. Since we feel that OpenDoc and CORBA standards are still evolving we should limit our work to full support of OLE 2.

One approach follows a focused approach in terms of market coverage. We don’t provide in the first release a broad range of functionality and platforms. Rather we concentrate on “Microsoft” platform and provide:

1. A novel interactive graphical development environment which includes GUI and relational DB access tools;

2. Very good integration: full C++ FFI;

Another approach would be to provide a traditional environment, but try to cover multiple platforms and broad functionality. Portability would be a very important element of this strategy.

3.2 Platform

We aim at Chicago and Windows NT. Hence portability layers are of lesser priority. Instead we interface well with “Microsoft platform”. That’s the approach we favour now.

3.3 Market

We target Visual C++ and Smalltalk markets. We compete with Smalltalk by having better delivery and interoperability story, and with Visual C++ by supporting dynamic development environment with rapid prototyping and better time to market times.

We will aim initially at the financial and general MIS custom development in small teams (2-5). The environment may be also attractive to certain shrink-wrapped applications.

In this approach we want to levarage the dynamic nature of our environment. If we concentrate on the graphical, directly maniupulated development environment, we will make the Dylan language accessible to developers with a broad range of skill levels.

3.4 User

The user typically will be either novice or experienced MIS programmer with exposure to Visual C++, Visual Basic, COBOL, maybe Smalltalk. We will not support large team programming.

3.5 Development Environment

The development environment is very graphical and interactive. It features a set of graphical browsers, an editor, interactive debugger, and a set of tools for non-programmatic development: GUI builder, schema designer.

3.6 GUI builder

We interface with a GUI builder. This might be off the shelf one like AppStudio from Visual C++ (this assumes that the user already has Visual C++), or we might OEM one with source and deliver with the environment. One such tool from Soft Bridge includes Basic interpreter.

3.7 Editor

We build our own editor (though we might use an emacs as a base) but we don’t interface with commercial editors. This is because the editor is an intrinsic part of the environment and the avail-

able mechanisms (like OLE 2) are not satisfactory for interfacing with third party editors. We provide bindings for various editors though.

3.8 Database

At this point we are planning to use ObjectStore. It would probably be beneficial to be able to interface to any PC based database, through ODBC.

3.9 Interoperability of applications

We interface with OLE 2. We don't plan in the first release to integrate with OpenDoc D/SOM, CORBA. We will support OCX's. Dylan will be able to deliver libraries (DLLs).

Our FFI provides very good interface to C++. We will support subclassing of C++ classes, specializing on C++ classes. We will also support exporting Dylan libraries as C++ interfaces.

3.10 Libraries

The product will contain the following libraries or interfaces to libraries: ODBC, UI layer (Windows 32), threads.

3.11 Platforms

3.11.1 For Developing Dylan Programs

- Minimum processor speed is 486DX4@ 66 Mhz
- Minimum RAM 24 to 32 Mbytes
- Minimum disk is 500 Mbytes
- Screen size should be 1024 x 768 x 8 (color)
- Pointing device and keyboard
- OS uses Win32 API (both Chicago and Windows/NT)

3.11.2 For Running Dylan Applications on Desktops

- Minimum processor speed is 486DX4@ 66 Mhz
- Minimum RAM 8 Mbytes
- Minimum disk is 100 Mbytes
- Screen size should be 680 x 480 x 1
- Pointing device and keyboard
- OS uses Win32 API (both Chicago and Windows/NT)

3.11.3 For Running Dylan Applications on Embedded Computers

The requirements for embedded platforms are significantly less in terms of processor speed, amount of RAM, and peripherals than they are for PCs. Our goal is to make such applications able to run on very small machines.

4. Models

4.1 The Market

Our initial target market is the set of programmers who are using Chicago and Windows/NT. The majority of these people are programming in C++ or Basic, using tools such as Visual C++ or Visual Basic. It appears to be the case that the numbers of such programmers is growing.

The popularity of the Power Builder and similar environments seems to be growing, too.

In this market, there is a growing number of "standards" being widely used or developed, including OLE (and OpenDoc), COM (and CORBA/DSOM).

By attempting to enter this market, we are operating on the assumption that Dylan and DylanWorks will offer programmers a better way of doing things than they have with their current languages and tools. However, it is clear that this alone is not enough to cause a project to switch to Dylan. First of all, the choice of language is frequently a management decision based on both technical and non-technical arguments. More importantly, the features of a programming language are only a part of the overall picture of productivity. Availability of various libraries and tools, the programmer's own experience, and interoperability with other systems all play a major role.

OO programming is already a prevailing paradigm in the markets we are targeting. We expect that many programmers in this community will be looking for a new language and environement that better meets their requirements than either C++ or VisualBasic. In the dyanmic object-oriented corner, we expect that Dylan and Smalltalk will be competing for this market, but only if they meet the other non-language requirements.

We therefore need to emphasize the things — both in the language and in the environment — in DylanWorks that better meet a programmer's needs than any other programming language and environment: dynamic nature of the language and the graphical style of the environment.

We can safely assume that Dylan programmers shouldn't be forced to rely exclusively on Dylan for any aspect of their work. To the contrary, they should be able to freely mix and match Dylan with other pieces of software.

4.1.1 Competition

The main competition in this market comes from C++ (probably using Visual C++), and there will probably be competition from some Smalltalk vendors. Depending on the type of user, there may

also be competition from Visual Basic. There also may be competition from Power Builder and similar such systems (GUI builder + database access + scripting + report generation).

This competition sets a certain baseline constraint: people need to be able to use DylanWorks at least as effectively as expert users now use Visual C++ or Visual Basic. Dylan applications need to integrate at least as well as applications written using those tools.

4.1.2 Legacy Software

Some organizations have [vast] amounts of legacy software. Do we intend to address such legacy code? Will DylanWorks directly address the needs of people who have a huge old system that they want to convert to Dylan? What if the old system is written in C/C++? What if it is written in Lisp? What if they just want to extend a legacy system? Is it enough to use FFI to interface to old code, or will there be a requirement that the tools supplied by DylanWorks can be turned loose on legacy code?

4.2 Users

In the “Types of Users” section on page 13 and “Organizations” section on page 15 we list types of users and types of organizational contexts (limited to team sizes at this point). To achieve broad appeal we will have to assume that the users are not “high end” programmers with experience with dynamic functional languages. We have to “enable” programmers with various skill levels. To address the needs of those users we need to provide intergrated tools (GUI, DB access) and very graphical and integrated and intuitive environment.

We will not be able to address the problem of large project management in the first release. Our database based environment will provide some support for team programming for teams of 2-5 people.

4.3 Applications

In the “Applications” section on page 16 we list categories of applications which DylanWorks might be used for. The custom application market of financial and traditional MIS applications is very attractive and Dylan is well suited for it. We want to levarage the dynamic nature of the language itself and the incremental style of development in our environment. The programmers in those organizations fit the model described above. Interoperability will be a strong point in this market when competing with Smalltalk. In future releases we would have to provide support for large scale project development.

5. Feature Space

5.1 Base Requirements

To get into the market at all, we need to perform well in the following areas:

- Dylan has to feel like a true, general purpose programming language.
- Delivered applications must have a suitably small footprint.
- Delivered applications should have naturally good performance. The programmer shouldn't be required to recode any part of an application in C to get acceptable performance.
- Ability to deliver linkable libraries that can be integrated with other libraries written in other languages (C, C++, Visual Basic) to form an application.
- We need to support the creation of OCX's.
- Dylan applications need to be as indistinguishable from applications written in standard languages like Visual Basic and Visual C++ from the standpoint of end-users and other programs that might communicate with them.
- Availability, either in Dylan or through other languages, of a rich set of libraries and components, including UI, graphics, database access, etc.
- Must support both mouse and keyboard for majority of operations.
- Reliability.

In order to really compete, we need to do exceptionally well in the following areas:

- Powerful, productive programming environment for developing complex systems.
- Support for both small and efficient applications, and for large complex systems.
- Dynamic language that supports true incremental development.
- Automatic memory management.
- Flexible, customizable programming environment which empowers an experienced user.
- Easy to learn environment which, with the on-line help, allows a novice user to be productive after even a single day. Inspires confidence in user abilities from the beginning.
- Fun to use (interesting, unusual).

It is also necessary to do the following well, but it probably suffices to interface to off-the-shelf tools that provide this functionality:

- GUI builder.
- Mail reader.

5.2 Functionality vs. Interoperability

There is a tension between being "standard" and breaking new ground. That is, in order to achieve a high level of functionality, a piece of software may be perceived as non-standard or even less than fully interoperable. This "functionality *vs*. interoperability" trade-off needs to be considered carefully.

We believe there are three goals in the area of functionality and interoperability; these goals apply both to DylanWorks and to software built using DylanWorks, although the exactly what the

requirements are for meeting the goals may be different for DylanWorks and for Dylan components. The goals are:

- Achieving a minimum level of functionality and a minimum level interoperability. These cannot be traded off against each other.
- Achieving additional functionality.
- Achieving additional interoperability.

Failure to meet the minimum level of functionality and interoperability is a “killer” for both DylanWorks and Dylan components. What needs to be done is to define the minimum levels for both of these goals, both for DylanWorks and for Dylan components. In contrast, the second and third goals may be traded off against each other, and the trade-offs may vary considerably from one component to another. In particular, the trade-off may be much different for DylanWorks than it is for any components produced by DylanWorks.

Note well that the interoperability requirement might put some limitations on the functionality we can deliver. However, the minimum levels of both must be met, and the remaining question is, how much can we compromise additional functionality to achieve additional interoperability (or vice-versa)? These compromises might be driven both by available resources, and by technical conflicts between functionality and interoperability.

5.2.1 Requirements for DylanWorks

Ideally, DylanWorks should be designed in a modular fashion so that programmers can use the tools they are used to. There are compromises to be made here. On the one hand, our own tools will have better access to the information captured by the environment, and as such may provide better functionality. Thus, integrating with third party tools may mean some degradation of functionality. On the other hand, there are some standard tools that people become very attached to; allowing integration with these tools can provide substantial benefit to DylanWorks users.

To support “external” tools, we will need to have a solid, published protocol that describes how the DylanWorks tools communicate with each other. This would allow us, and others, to integrate external tools with DylanWorks. The protocol should be designed to enable graceful degradation if external tools are not as functional as they might be. But it must be remembered that this approach can be quite costly to us in terms of the amount of design, implementation, and documentation work we have to do. It is more work still if we ourselves do the work of “plugging in” a lot of external tools.

It is certainly the case that we do not have the resources to compete with every potentially useful tool. But some third party tools will provide more functionality than we could deliver directly ourselves. In particular, a GUI builder is one tool that we can productively integrate with.

5.2.2 Requirements for Dylan Applications

1. We must be able to interoperate with other components on equal terms.

We want to be able to make Dylan components that can be used with other components which may or may not be written in Dylan. In no sense can we make any assumptions that the Dylan components are “in charge” of any operating system resources (such as memory, threads, event queues, etc.). Equally, we cannot assume that Dylan components get control before other components, since Dylan may be called from other components. We cannot expect the other components to be recompiled or relinked. Nor can we expect to limit their use of OS API calls.

2. We must not prevent the sharing of operating system resources.

We should not make any assumptions that operating system resources might be limited by component boundaries. For example, any component can create a thread which may then execute within other components. Threads created in Dylan must be usable by foreign components and vice-versa.

Again, there is a trade-off, this time involving efficiency as well. We to produce applications that are interoperable, but they also need to operate efficiently. On the one hand, applications will be judged in large part based on how well they “fit in”. On the other hand, if the applications happen to work poorly because some part of a Dylan library is slow or inefficient, then the application will still judged unfit.

5.2.3 Specific Issues

Here are some specific places where we need to trade off interoperability with functionality and efficiency:

- Use of local window system API (such as MFC) *vs*. a higher level UIMS.
- In general, use of local API *vs*. more portable Dylan bindings to various bits of useful functionality.
- Threads *vs*. implementation of the memory management module.
- The FFI and Creole in general. We need to integrate smoothly with C and C++ for call-in, call-out, subclassing C++ in Dylan, and maybe even subclassing Dylan in C++.
- OpenDoc and Ole, and how they fit in with the UIMS.
- CORBA/SOM, and COM.
- We need to be able to produce OCX’s.
- We need to be able to produce DLL’s.
- DylanWorks should integrate with a good, off-the-shelf GUI builder.

5.3 Portability

If we can do a good job at it, a good portability solution could well be a strong selling point of Dylan. It is not the case that portability must necessarily compromise integration, but we need to be very careful that the quest for portability does not compromise interoperability, especially in the area of the user interface of delivered applications.

Note that we again have to concern ourselves with the portability of both DylanWorks and Dylan components. These need to be considered separately.

We think that making DylanWorks as portable as possible may be a good move in the long run. We plan to try to make DylanWorks portable, but the highest priority is the quality of the product on the Windows platform.

We also plan to make Dylan applications as portable as possible. The principle way of doing this is to ensure that we develop API's for libraries that will themselves be portable. Again, it is not the highest priority to port these libraries to every conceivable platform, but it is a priority to cooperate with other organizations who might do this. That is, it is an explicit goal to standardize on core libraries across different Dylan implementations, such as Apple's and CMU's.

5.4 Style

Software environments can range in style from the mostly text-based (Gnu Emacs and gcc), through hybrid environments (like LispWorks or Visual Basic), to more pure visual languages (like Prograph).

We intend to make DylanWorks be a hybrid environment. Many of the browsing tools will be very graphical in nature, but will often "bottom out" in a textual source code editor. There will be a graphical GUI builder (either bought or built), but when the GUI is connected to the rest of the application, it will probably also be done by editing program source text. The key here is to use the style appropriate to the task.

5.5 User Interface Substrate

We are presently planning a "three pronged" strategy for dealing with the User Interface substrate.

This is described in more detail in the UI Substrate white paper, but the summary is this:

- There will be a non-portable low-level Dylan binding to the native UI tools, for instance, a binding to the Win32 UI tools.
- There will a portable, higher-level layer built on top of this. This layer corresponds roughly to CAPI, the Silica layer of CLIM, or to Fresco, and implements a fairly traditional events and gadgets toolkit (although it will also support OpenDoc-like embedding).
- There will be a portable, very high level layer built on top of that. This layer corresponds to the presentation-based style of CLIM.

Interfacing to non-standard, third-party widgets will be provided at the lowest level of the UI substrate. If some of these widgets become widely used, then they can be supported at the higher layers of the UI substrate as well.

The standard GUI toolkits typically provide only a base UI functionality. Some applications need to develop custom interactive user interface elements, such as graphers. Having a powerful tool (CLIM-like) that accommodates this need and seamlessly integrates with the native toolkits would be a strong selling point of our environment.

5.5.1 GUI Builder

There is also the issue of a GUI builder. Most people are used to building GUI's in a compositional, modular fashion. There are good reasons for people to use GUI builders. Some interfaces can be created faster and in an interactive way. This is often better than doing the same thing programmatically,and is almost always better than trying to do "programming" by hacking resources. It is also the case that non-programmers can design a UI using a GUI builder.

For the initial release of DylanWorks, we plan to interface our own UI substrate with one or more GUI tool available on the market, since building our own GUI builder is not trivial. An open issue here is how to interface the output of a third-party GUI builder to the high-level layers of the UI substrate (for example, CLIM commands).

Later releases of DylanWorks may have a GUI builder written by us that supports building interfaces that use the higher level functionality provided by CLIM.

5.6 Editors

The editor for DylanWorks is a difficult issue. On the one hand, it is a very important tool for a developer and frequently reflects strong personal preferences. On the other hand, it is a central part of any development environment.

We would like to add some sophisticated features to our editor, but unfortunately there are no standard ways to interface with most editors. The OpenDoc/Ole framework may eventually be sufficient, but we do not know when (and we are not sure). So, although it may be beneficial to do so, it is not clear how well we can interface with third-party editors.

Even assuming we implement our own editor, we do plan to provide different sets of key bindings to accommodate users of different editors.

5.7 Other functionality areas

There are other places where functionality versus interoperability (and often, buy versus build) comes up. These issues are addressed in the other white papers.

- Source version control and system configuration management tools.
- Debuggers.
- On-line help (that is, we should use Microsoft Help system).
- Ability to use local resource model.
- Relational database access library.
- Object-oriented database access library.
- 3d graphics, such as Open GL.
- Multi-media, such as QuickTime.

Appendix A. Types of Users

When describing potential DylanWorks users, we considered the following questions:

- What programming languages and environments have they used (C/C++, Visual Basic, Lisp, Smalltalk)? Since Dylan is new, we can assume that most such users don't already know Dylan.
- What do they know about object-oriented programming?
- What operating systems have they used? On what platforms?
- How many years of programming experience?
- What is their capacity to understanding large complex systems?

A.1 High-school Student

- High-schools students are not likely to know much of anything about writing software.
- Shorter attention span than other types of users.
- Will not be writing complex software.
- Would like DylanWorks to be fun to use.
- Would like to do things with a flashy user interfaces and multi-media.

A.2 College or Grad Student

- Will probably already know one or two programming languages.
- Short attention span due to pressure from other courses.
- Will not write complex software, unless it's a thesis.

A.3 Novice Professional Programmer (fresh out of college with CS degree)

- Knows algorithms and C, but not Dylan or advanced programming environments.
- Does not know real-world software engineering techniques. Might or might not know anything about object-oriented programming.
- Will be starting to write complex systems.

A.4 Experienced Basement Hacker

- Probably knows C/C++ and Basic
- Has experience mostly on Windows or the Mac.
- Has probably used something like Visual C++ or Visual Basic.
- Works alone.
- Strongly motivated, probably willing to learn new things.

- Doesn’t have much money to spend.
- Small applications, often distributed as shareware.
- Definitely wants the language and environment to be fun to use.
- Probably wants to be able to write cool-looking programs.

A.5 Experienced Professional Developer (C/C++)

- Knows C/C++, but unfamiliar with dynamic languages.
- Significant fraction probably have a Unix background.
- Older such developers may have never used any real programming environment, but younger ones probably will have.
- Works with teams. Size of team can vary widely.
- Not willing to spend lots of time learning, especially completely new ideas.
- Medium to large applications.

A.6 Experienced Professional Developer (VB)

- Knows Visual Basic, but unfamiliar with dynamic languages.
- Small teams, or works alone?
- Maybe willing to spend some time learning new things.
- Small to medium applications?
- Used to simple languages and a very structured model of program development.

A.7 Experienced Professional Developer (Smalltalk, Lisp)

- Knows object-oriented techniques and dynamic languages very well.
- Used to working with large software systems.
- Not necessarily good at delivering a tight, finished product.
- Works alone or in teams.
- Unemployed, hence receptive to new ideas.
- Wants the environment to be fun.

A.8 Experienced Professional Developer (Cobol)

- Unfamiliar with object oriented techniques.
- Unfamiliar with GUI.
- Unfamiliar with dynamic programming environments.
- Has a lot of legacy code.

A.9 Engineering Professional or Scientist (Fortran)

- Unfamiliar with modern languages.
- Needs access to extensive, high-performance math libraries.
- May have a lot of legacy code.

A.10 Domain Expert (not a computer scientist)

- Spends a lot of time programming, but their real expertise and training is in some other field.
- Works alone or in small teams.
- Needs to be able to build software that does a domain-specific task well.
- Needs to be able to program “experimentally”.
- Software is often highly complex, since it reflects some complex real-world structure.

A.11 Commercial project manager

- Very conservative.
- Risk averse.
- Concern for software maintenance, customer support both from Harlequin, and how he will support his own customers.

Appendix B. Organizations

In the following sections, we describe the needs that are of particular importance to a kind of user or organization.

B.1 Single User

- Needs tools that highly leverage an individual.
- No need for groupware.
- Will need organizational tools to help maintain software over time.

B.2 Classroom (programming)

- Good on-line help and tutorial.
- Novice-friendly environment.
- Remote execution and debugging (for the teacher to control student’s workstation).

B.3 Small Team (2-5)

- Needs tools that highly leverage an individual.

- Some need for groupware.

B.4 Medium Team (5-20)

- Groupware is more important than highest possible productivity for individuals.

B.5 Large Team (20-100)

Note: Release 1 of Dylanworks will not support large teams.

- Groupware is the most important consideration.
- Support for formal procedures: baseline and release builds.
- ISO-9000 support?

Appendix C. Applications

C.1 Custom In-house Corporate Applications

Financial

- Might be “soft” real time.
- PC and UNIX.
- Numerical.
- Relational and OODB access.
- Presentational graphics (pie charts).
- Constantly ongoing development to meet changing requirements.

Engineering

- Might be “hard” real time.
- UNIX.
- Might be numerical.
- 3-d graphics (CAD systems and visualization).

General MIS

- PC plus server (mainframe, too?).
- Relational DB access.
- Form filling interfaces.
- Printing
- Software is frequently modified to meet new specifications.

C.2 Contract Jobs

- Integration
- Networking

C.3 Shrink-wrapped Applications

All of these applications require the following:

- Customer support utilities: patching, bug reporting, bug tracking and remote debugging utilities.
- Communication paths between end user and the software vendor.
- Portability (Macs and PCs).
- Maintenance tools for keeping customers up-to-date.

Games

- Fast.
- Small (more or less).
- Multimedia, animation.
- Portability (Macs and PCs).

Device Drivers/Embedded Controllers/ Low Level Tools

- Efficient access to low level OS functionality.

Complex Tools and Utilities (data bases, programming environments, etc.)

This kind of projects can often be characterized as complex systems that have evolving requirements because the problem itself is not well understood (yet).

- Team development.
- Version control.
- System configuration management and build utilities.
- Fast compile times, incremental compilation and dynamic linking.
- Efficient compiled code in the final application.
- Portability?
- Support for browsing complex software structures is critical.

Others (word processors, spreadsheets, etc.)

- UI intensive.
- Portability (Macs and PCs).

At Harlequin created on September 23, 1994 and last modified at Harlequin on November 11, 1994.