TIDE: Timeliner Integrated Development Environment
M. Eng. Thesis Proposal

Maya Dobuzhskaya
mdob@mit.edu
August 13, 2004

Abstract
The Timeliner system is used for real-time control on the International Space Station. The current method of Timeliner script development has usability drawbacks and would benefit from a coherent development environment. This paper proposes TIDE, an integrated development environment built as a plug-in to the Eclipse Platform. TIDE will incorporate existing Timeliner stand-alone tools, and will contribute editing and search functionality. It will include an editor with custom language definitions linked with a real-time compiler, and an interface to a database, with browsing and powerful search capabilities. TIDE will be extensible, and will provide good usability for both novice and expert script developers.
1 Introduction

1.1 Timeliner Background

The Timeliner system was developed in 1982 by Draper Laboratory to automate procedural tasks. The first release was used to emulate the timelines of on-board crew procedures on the Space Shuttle. The Timeliner system is both a computer language and an execution environment. The language is used to write scripts that provide sequencing and process control, and the execution environment provides real-time monitoring and control [29].

Timeliner allows scripts to be developed and executed in system environments that are linked to a database. Scripts can be written for space applications such as autonomous or interactive vehicle control, performance of preflight subsystem checkouts, handling failure detection and recovery, or stepping through pre-defined procedures of a scientific experiment. Other applications include process control for manufacturing, materials processing, robotics, or any automated procedures that need to be executed reliably. Currently, Timeliner is used on the International Space Station’s real-time core command and control, and payload control computers [29].

The current Timeliner script development process involves composition and compilation. A Timeliner script, called a bundle, is composed in a text editor. Scripts contain one or more sequences, sequential sets of actions that contain instructions for monitoring, controlling, and reporting on the operations of a target system. Each script is linked to a database where commands and values are stored. Figure 1 shows a sample Timeliner script [14]. These scripts are then provided to a compiler as ASCII text files. The compiler checks language syntax, looks up references in the database, and produces an executable [29].

1.2 Motivation

This script development process has three major drawbacks. First, a script writer must know what commands and variables (called telemetry data) are available for inclusion. These values are specific to a database, and since each database can contain thousands of values, it is often tedious searching through files to find what is needed. Second, a generic text editor provides no language specific assistance such as keyword highlighting or auto-indentation. This makes scripts prone to syntactic mistakes [17], and makes it difficult for novice script writers, since they have to look up or memorize Timeliner keywords. Lastly, usability of the Timeliner system could be greatly enhanced if existing tools were integrated into one coherent environment. For example, script creation would be more efficient if the compiler could be invoked from the editor, and the results could be displayed right in the editor [17].

1.3 Goals

This paper proposes a Timeliner Integrated Development Environment (TIDE) that would address all the drawbacks mentioned above, and allow for any person to easily and quickly develop a Timeliner script. For TIDE to be useful, it should have the following features:

- **Usability.** The environment and tools should be easy to learn and use. A consistent user interface should be presented.
- **Extensibility.** The environment should easily accommodate additional tools and changes in the Timeliner language.
The rest of this paper is structured as follows: Section 2 describes related work in terms of languages similar to Timeliner, brings up text editing issues, and introduces development environments. TIDE’s overall design is presented in section 3, followed by implementation details in section 4. Section 5 suggests a strategy to evaluate TIDE’s usability. Finally, a proposed implementation schedule is given in section 6.

2 Related Work

This section describes languages similar to Timeliner, reviews text editing advances, and examines development environments.

2.1 Other Real-time Control Languages

Similar to Timeliner, the Spacecraft Command Language (SCL) is used for controlling events. SCL is a collection or components that are integrated through a database. It was originally developed as an embedded software system to operate a U.S. Navy satellite for 180 days without ground intervention [22]. Unlike Timeliner, it is not deterministic, since it can respond to events in a rule-based, as well as a time-based manner. Furthermore, SCL is an independently running platform,
while Timeliner is meant to plug into a platform.

Also comparable to Timeliner, Supervisory Control and Data Acquisition (SCADA) Systems have been in use since the 1960s for gathering and analyzing real-time data. They can monitor and control a plant or equipment in industries such as telecommunications, water and waste control, energy, oil and gas refining, and transportation [24]. Unlike Timeliner, contemporary SCADA systems exhibit predominantly open-loop control characteristics and utilize long distance communications [25].

2.2 Language-based Text Editors

Since computer languages are highly structured and have clearly defined syntax, editors should not treat them as plain text [17]. Language-based editors exploit their knowledge of the syntax and semantics of a given language to assist the user in forming correct expressions [6]. Such editors provide better usability since they give immediate feedback to the user through real-time assistance such as syntax highlighting and auto-indentation [17]. In the 1980s when these language-specific editors were emerging, a question arose about how much discipline to impose on program construction [6].

The choice was between a syntactic approach or a lexical approach. The syntactic approach uses formal grammar and parses text into a tree of well-formed constructs. While this makes code more expressive, it limits the user’s freedom to pass through inconsistent intermediate states, since the approach forbids ungrammatical code. The lexical approach is less formal, and treats text as a sequence of flat segments. While this approach gives users complete freedom, it doesn’t take advantage of language constructs to guarantee syntactic consistency [20]. Since users switch between perceptions of symbol sequences and character sequences with little conscious effort, a compromise between the two approaches must be made [6]. The Cornell Synthesizer Generator [28] is a hybrid between the two approaches. It parses text incrementally, allowing users to freely edit code which it subsequently parses into tree nodes [20].

Another language-based customization in editors is content assist. When a user types characters, the editor presents possible completions for the word based on known keywords. This technique dates back as far as 1981, when it was used in a study of tactical data entry methods [9]. Today, content assist is used in development environments to make typing more efficient.

2.3 Development Environments

This section outlines the history of development environments, depicts the advantages and problems of modern Integrated Development Environments (IDEs), and explains how the Eclipse Platform [7] overcomes these shortcomings.

2.3.1 History

Software engineers use environments with tools such as editors and compilers to develop code. Earlier environments, such as Unix [16], loosely interconnected the inputs and outputs of tools via redirection. However, these early environments didn’t provide for a way to integrate tools, coordinate their activities, or automate common tasks. The earliest vehicles for automating the flow of control among tools provided for a way to describe how tools and data relate, and which tools should be used to process changed data. These vehicles, such as Make [10], enhanced the usability of stand-alone tools, but the resulting environments still only provided loose tool integration [12].
The first significant efforts in producing tightly integrated development environments were Programming Support Environments (PSEs), collections of tools that support coding activities. Earlier PSEs, such as the Cornell Synthesizer Generator, typically provided language-specific editors, compilers, and debuggers. These tools were tightly integrated so that the activities of one tool were reflected appropriately in the other tools. PSEs were extremely useful applications, but were limited because they could only support one software engineering activity and its artifact, or implementation and code [12].

Software Engineering Environments (SEE) emerged from the need for integrated support of software engineering activities throughout the software lifecycle [12]. For example, Arcadia SEE [1] included tools for requirements specifications, analysis, and testing. From SEEs, two important lines of research emerged: Multi-view Software Environments, and Process-centered Software Engineering Environments (PSEEs). The former allowed multiple views of a given piece of software where a modification to one view would cause the other views to be automatically updated. The later combined tool support for software artifact development with support for the modeling and execution of the software engineering process that produced those artifacts [12].

2.3.2 Integrated Development Environments

Modern Integrated Development Environments (IDEs) stemmed from PSEEs, and include the explicit representation of processes, their products, and their interactions [12]. Early IDEs were highly specific to some content. However, popular IDEs in use today such as Emacs [27], Microsoft Visual Studio [19], and the Eclipse Platform allow for adaptation and integration. For instance, they provide support for several different programming languages.

The two biggest criticisms of current IDEs are that they are too large and too complicated to use. Since each IDE tries to bundle together all the possible tools a user might need, some critics claim IDEs are swelling to large proportions [4]. A solution is to modularize IDEs and have them provide support for independent integratable plug-and-play tools which can be reused within multiple IDE frameworks [4, 12]. Other critics allege that since IDEs encompass many tools, they might be daunting for novices. However, if the user interface is well designed, IDEs can be simple to use for both experts and beginners.

2.3.3 Eclipse: A Modular and Extensible IDE

The Eclipse Platform, an open-source workbench for the integration of software development tools, elegantly meets the aforementioned modularity goal that other IDEs lack. Eclipse is entirely built from modules, called plug-ins, allowing for the seamless integration of independently developed components. Plug-ins range from ones essential to the Eclipse workbench such as core, ui, and swt, to more specialized and smaller ones such as the JUnit testing plug-in. Each plug-in consists of source code and an XML [5] manifest file. On startup, Eclipse reads through all the plug-in manifest files and hooks up the appropriate modules into the system.

Eclipse facilitates the construction and integration of new plug-ins through well-defined Application Programming Interface (API) interfaces and a Plug-in Development module that provides useful building blocks and frameworks [7]. Currently, web directories list 612 existing Eclipse plug-ins in varying stages of completion, of which about 400 have been created within the past year [8].

Eclipse’s modular and extensible nature can be attributed to four features: explicit extension points, a multi-level architecture, self-description, and component encapsulation. First, Eclipse has
distinct ports to which other plug-ins can connect to as long as they adhere to a given interface. These explicit extension points make it easy to see where and how components can be plugged into the platform. Second, Eclipse plug-ins can declare their own extension points, allowing the tool architecture to have multiple levels. Third, plug-ins provide manifest files to allow the platform to be aware of the essential properties of a plug-in before its source code is loaded. Lastly, Eclipse effectively encapsulates plug-ins. A plug-in can access another plug-in only if it is declared “required” in the manifest. Each plug-in has a class loader, and runs in a container of its own that regulates communication with other components to avoid unintended dependencies between plug-ins [18].

3 Design

This section gives an overview of TIDE, and explains how it will meet the usability and extensibility goals set forth in the introduction.

3.1 Overview

TIDE will be built as a collection of Eclipse plug-in using the Model-View-Controller (MVC) [11] design pattern. Eclipse was chosen because it is open-source, modular, extensible, and has a good user interface backbone. The MVC design will be used to separate the user interface from the handling of text, searching of the database, and compiling.

Classes will be divided into model classes, view classes, and controller classes. The model classes will perform the actual functionality. They will handle document editing, provide database search capabilities, and perform incremental compilation. The view classes will display information to the user. They will include an outline, navigator, search, reference, and tools views. The controller classes will handle user actions. They will respond to the platform’s control methods invoked from the editor, launch Timeliner tools which have been integrated into the platform, and handle events associated with the views and search functionality.

3.2 Usability

TIDE will be easy to learn and use for all user levels from beginning users to experienced programmers [21, 26]. Each of the user capabilities levels has its own traits, and dictates how users will interact with the environment [26]. To accommodate all users, TIDE will target the lowest capabilities level by providing a simple interface, while allowing environment customizations for advanced users.

TIDE will also present a consistent user interface. A consistent interface provides the user with a familiar look-and-feel for all constituents of the environment [2]. TIDE will meet the consistency goal by using the Eclipse SWT package for all the views, and by sharing common commands with general text editors.

3.2.1 User Interface Design Process

The user interface design process begins with task analysis [13]. I interviewed several people who produce Timeliner scripts to identify and understand their goals and tasks, what problems
they experience, and the changes they would like to see in their tools. From these interviews, I constructed a list of desired features for TIDE.

Following the Spiral Model [3] of iterative software design, the next step is a low-cost paper prototype. Concentrating on the layout of IDE pieces, namely views and buttons, I constructed a diagram of a potential user interface for TIDE (see Figure 2) and presented it to several script developers to get feedback.

![Figure 2: TIDE paper prototype.](image)

Next, I will implement a front-end Graphical User Interface (GUI) for TIDE with some core back-end functionality. The initial functional prototype will have:

- an editor with syntax highlighting,
- a perspective defining all the views,
- a ground database panel with simple search functionality, and
- a way to launch existing Timeliner tools.

This prototype will give a preview of the ways in which TIDE will overcome all the shortcomings of the former script development process. Namely, the prototype will demonstrate a simple interface
to view and find database values, an editor specific to Timeliner, and the integration of Timeliner tools. The main goal of this prototype will be to get feedback on the ground database panel, as that will be a feature specific to TIDE, and not a standard part of IDEs.

Taking into account feedback from this prototype, I will continue to construct the back-end functionality. Finally, the fully operational TIDE version will undergo a second phase of user testing before being ready for release.

3.2.2 TIDE User Interface

The TIDE user interface will be modeled after the Java Development Tools (JDT) Eclipse plug-in in terms of the layout perspective and editing functionality. As shown in the paper prototype (see Figure 2) the layout consists of an editor in the center, surrounded by navigator, outline, reference, compiler error, and database views. The database view consists of an area to enter a search query, and areas for displaying command and telemetry data. Editing functionality will include standard editing strategies such as syntax highlighting and auto-indentation, as well as smart editing features such as content assist. These editing features will benefit users by minimizing the amount of typing necessary, and by presenting nicely formatted text.

3.3 Extensibility

TIDE will easily accommodate additional tools [23], and changes in the Timeliner language. Since TIDE builds on the Eclipse framework, smooth tool integration is automatically guaranteed. Adding an extra tool or feature to TIDE simply involves implementing a well-documented API interface. Similarly, TIDE will be able to accommodate changes in the Timeliner language by having a single interface that handles language keywords.

4 Implementation

Like all Eclipse plug-ins, the TIDE plug-ins will consist of XML manifest files and source code. The manifest files will declare that TIDE depends on the Eclipse core and ui modules. One manifest file will state that TIDE implements extensions for an editor, views, actions, a file wizard, and a perspective. The other manifest file will define the Timeliner compiler that will run inside Eclipse. All source code will be written in Java [15].

4.1 Editor

TimelinerEditor will implement Eclipse’s ITextEditor by extending TextEditor. TextEditor provides a lot of the basic editing functionality such as copy, cut, paste, and undo actions, as well as an ability to save and load files. TimelinerEditor will have a custom document provider, source viewer, and action contributor. The document provider will be used to partition the document for syntax highlighting. The source viewer will define smart editing features such as auto-indentation. The action contributor will implement editor actions such as content assist.

4.2 Views

The following TIDE views will each implement Eclipse’s IViewPart: navigator, outline, reference, compiler error, and database view. A navigator view is provided by Eclipse for navigating through
the open files in a TIDE project. An outline view will show all the sequences in a script and
will support jumping to a certain sequence by clicking on its name. The reference view will be
a tree viewer that contains Timeliner keywords sorted by category. Users will be able to browse
through this tree and click on a keyword to insert it into the editor. The compiler error view will
list problems with a brief description and line number. Clicking on a description will highlight the
questionable region of code in the editor.

The database view will allow users to browse and search for values in the database associated
with an open script. Two tree viewers will present commands and telemetry data with their
associated types and parameters. These viewers will be populated from a table that will be created
as a result of parsing database files. The back-end of the search functionality will be implemented
as the edit distance algorithm. The algorithm will provide spell checking, account for wildcards,
and ultimately return the closest match.

4.3 Actions

The existing Timeliner tools will be added to the Eclipse main toolbar and will implement Eclipse’s
IWorkbenchWindowActionDelegate. Buttons will be added to the toolbar that will launch GUIs
to create a database, compile a TLS file, view a TLX file, and start the test environment. Currently
these GUIs are implemented in Swing and run as stand-alone tools. Their integration into TIDE
will include a conversion to SWT.

4.4 Wizard

TimelinerNewFileWizard will implement Eclipse’s INewWizard. Once the user chooses to create a
new Timeliner script, a pop-up GUI will prompt him or her for the file name and query if template
code should be inserted into the new file.

4.5 Perspective

TidePerspective will implement Eclipse’s IPerspectiveFactory and will be modeled after Ec-
lipse’s JDT perspective. It will define menu contributions and the layout of the editor and views on
the workbench. The editor will be positioned in the center, surrounded by the navigator, outline,
and reference browsing views on the left, the compiler error view at the bottom, and the database
view on the right.

4.6 Compilation

Scripts in TIDE will be compiled with the Timeliner Java compiler. The compiler will be a plug-in
that will run in the same Java Virtual Machine as Eclipse. It will execute automatically whenever
a script is saved, and compilation will be incremental, only recompiling modified regions. Errors
will be underlined in the editor and presented with a description and line number in the compiler
error view.

5 Evaluation

To evaluate TIDE’s usability, I will conduct user studies observing people interacting with TIDE.
The user pool will consist of both people with a general coding background and people who normally
write Timeliner scripts. I will compare TIDE with the old method of script development, which consisted of writing a text file and launching appropriate tools. I will observe how easy it is for users to utilize each method based on their behavior and time it takes to complete a cycle of the script development process.

6 Schedule

- **Summer 2004**: Develop a functional prototype with basic editing features and views in place. Include simple search functionality.
- **September 2004**: Deploy the prototype and make changes based on initial user feedback.
- **October 2004 - March 2005**: Complete the back-end functionality. Parse the Timeliner database files and populate the database views, complete full database search functionality, implement real-time compiling, change toolbar GUIs to use SWT, add bells and whistles to the editor, and provide a help section.
- **April 2005 - May 2005**: Evaluate the product based on user studies and write the thesis paper.

References


[25] SCADA Stuff. Spread Spectrum Scene Online Magazine, 


[29] The Timeliner User Interface Language (UIL) System for the International Space Station. 