HARVEY MUDD
COLLEGE

Mathematics/Computer Science Clinic

Final Report for
*Community of ODE Educators (CODEE)*

# Redesigning *ODE Toolkit*

May 8, 2009

**Team Members**

Eric Doi (Project Manager)
Steven Ehrlich
Richard Mehlinger
Andres Perez

**Advisor**

Professor Chris Stone

**Liaison**

Professor Darryl Yong

# Abstract

This document details our redesign and refactoring of *ODE Toolkit* for the *CODEE* 2008-09 Computer Science/Mathematics Clinic. We outline the project as it stood when we found it, describing the architecture, giving a description of important classes, and explaining our decision to focus on refactoring the code. We then detail our work, including summaries of our new design, major classes, the motivations behind our decisions, documentation, improvements we made to the code, end results, and other deliverables.

# Contents

# List of Figures

# Acknowledgments

We would like to thank our liaison Professor Darryl Yong for his assistance and for his devotion to the *ODE Toolkit* project. We would also like to thank Martin Hunt '08 for taking the time to discuss with us the current state of the project and for providing much useful insight into its history and design. We would like to thank DruAnn Thomas for her constant work in making the clinic program run smoothly, and also Professor Bob Keller for leading the program. Finally, we would like to thank our advisor Professor Chris Stone, without whose invaluable assistance and guidance our work would not have been possible.

# Chapter 1

# Introduction

## 1.1 Background

*ODE Toolkit* is an educational program designed to solve Ordinary Differential Equations (ODEs). While there are a variety of other software programs available that solve differential equations, most are either expensive, general purpose mathematical tools (such as *Maple* and *MATLAB*) or are too specialized to be useful throughout an entire class.

While numerical solving of ODEs is not the focus of most introductory differential equations courses, properly designed software can assist instruction in a variety of ways. Software can quickly show the behavior of solutions, aid in developing the student's intuition, and show models useful to other disciplines. Software is also ideal for demonstrating how the behavior of simple systems is dependent on initial conditions and critical regions.

Ideally, good software can also show the limits of numerical solutions. With good examples, software can demonstrate where individual solvers fail. A good software package can show how different solvers deal with numerical issues and also compare numerical and analytical solutions to demonstrate the limits of numerical methodologies.

The Community of Ordinary Differential Equations Educators (CODEE) sponsored both the original *ODE Architect* and its replacement, *ODE Toolkit*. *Architect* had been used by the HMC Math Department for many years to teach differential equations courses. However, it is antiquated and proprietary, and only runs on Windows.

This is undesirable for several reasons. First, scientific and academic software should be open to peer review and enhancement. An open source

license would allow experts and enthusiasts to develop, improve, enhance and debug the software. Even better, those who work on open-source projects typically do so for free. In addition, many in the educational community do not use Windows. Mac OS X and Linux are increasingly popular, so it makes little sense to restrict the operating system to Windows when Java allows easy portability.

## 1.2 Initial State of *ODE Toolkit*

Prior work contracted by the Harvey Mudd Math Department produced software adequate in some respects but lacking in others. At the beginning of our project, the software was mostly functional. The software could accept systems of first-order ODEs as input, solve them, and graph the results. Version Alpha 0.9 was able to manipulate the viewing window, save and load sessions, and display additional information, such as direction fields.

However, we were concerned by the state of the internal structure of the code. Over the course of the past five years, work on the project was conducted by a series of Harvey Mudd undergraduates. The code base grew organically, often in non-obvious, convoluted ways. Due to the ad hoc nature of past development, documentation was nearly nonexistent and the code was unmaintainable.

## 1.3 Problem Definition

Our goal was to maintain all current functionality of *ODE Toolkit*, thoroughly document the existing code, and overhaul the architecture, implementing a more modular design. We intended to improve the maintainability and upgradeability of the *ODE Toolkit* software, while making adjustments and enhancements to the user interface experience. Figure 1.1 shows our final design on a phase plot tab. Our final version of *ODE Toolkit* is referred to as version alpha 1.0.

Future developers are our primary audience. We provided thorough documentation on the architecture and all components of the program. Developers will be able to rapidly familiarize themselves with the layout of the software, quickly determine any component's role, and easily improve the new modular architecture.

The biggest single issue that we saw in the original *ODE Toolkit* was that the code was minimally documented and poorly organized. Because

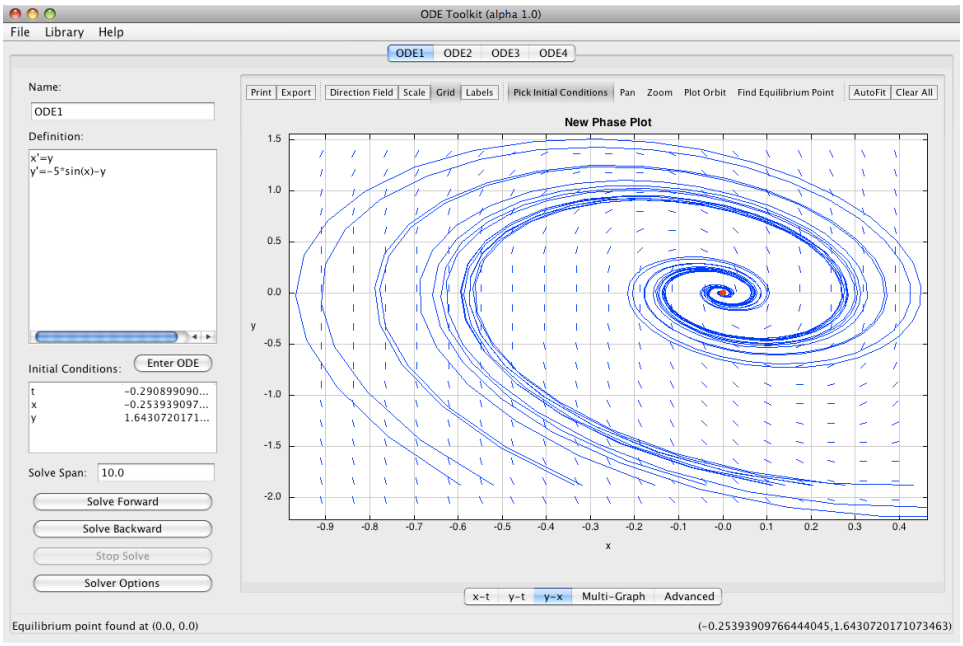


Figure 1.1: The *ODE Toolkit* version Alpha 1.0 user interface

the architecture developed over several years, with pieces implemented by numerous individual coders, the code had no central plan or organization. This led to an unwieldy code base that was unnecessarily difficult for new coders to maintain, upgrade, debug, and comprehend. Furthermore, the lack of consistent documentation discouraged new coders from documenting their own work. In its disorganized state, any further changes or upgrades made to the program could have actually exacerbated this problem by further complicating the architecture, ultimately leading to more frequent and more severe bugs.

Thus we decided to reorganize and refactor the code into a clear, modular, well-documented architecture. We completed the process of documenting the current architecture in the fall semester, and began refactoring the code to fit our architecture plans in the spring.

# Chapter 2

# Original Architecture

Before discussing the changes in our new design, it will be useful to explain the code's original architecture of the code. The majority of our refactoring work focused on the user interface package, so we will describe the most problematic details here. Figure 2.1 shows the role of the User Interface in the architecture. An overall summary of the remaining packages in the original architecture can be found in Appendix B.1.

## 2.1 User Interface

The User Interface package's name was very misleading. Although each class in the user interface served as an actual Java user interface component, the package was also responsible for various calculations, drawing graphs, and program control. The `main` function was found in GUI, the class that defines the highest level user interface component (the entire program window). The majority of the project's hand-written code was in this package, which was indicative of its inappropriately large scope.

This lack of modularity went down to the class level. It was not merely that the package included inappropriate files, but that its classes included inappropriate functionality, making maintenance and code comprehension difficult.

The following summarizes the most problematic subpackages and classes. Figure 2.2 identifies the classes defining the main parts of the User Interface.

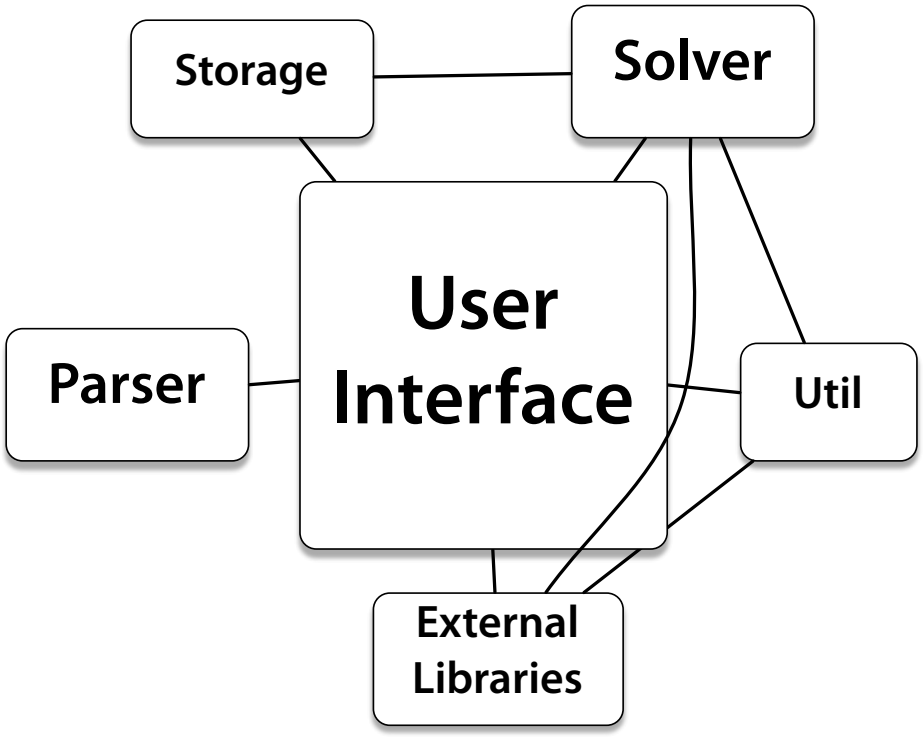


Figure 2.1: Diagram of the main components in the original architecture of *ODE Toolkit*.

### 2.1.1 GUI

GUI was the main Java window frame for the application, but also controlled the entire program. In fact, it housed the `main` function, which simply instantiated a GUI. The constructor handled the initialization of the program, loading external libraries and creating a new ODE file to work on. GUI held multiple open ODE files, in the form of ODEWorkspaces, in a tabbed panel.

### 2.1.2 ODEWorkspace

ODEWorkspace was the central container for ODE information, solver parameters and all other variables related to inputting, solving and displaying ODEs. It contained a panel for specifying ODEs and parameters and a panel for viewing the resulting solutions, as well as a status bar for useful

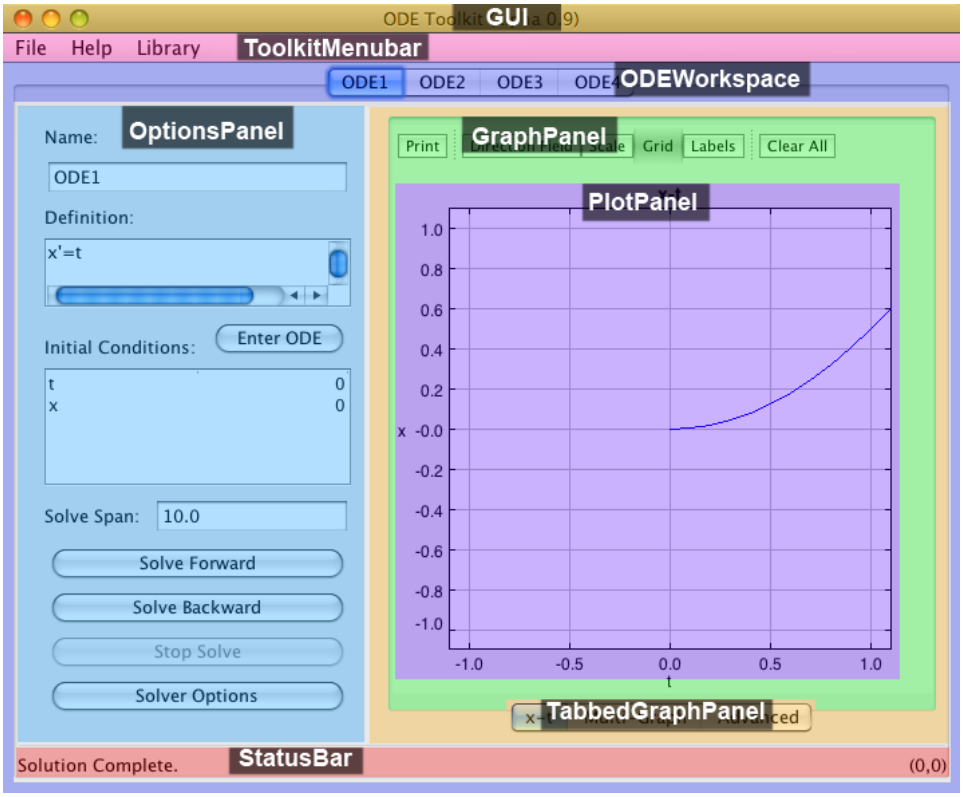


Figure 2.2: Screenshot highlighting the areas of the display controlled by each UI component.

information, typically where the user last clicked on the graph.

### 2.1.3 TabbedGraphPanel

TabbedGraphPanel contained a number of GraphPanels and mediated between them and the ODEWorkspace. Each TabbedGraphPanel created ComponentGraphPanels for each dependent variable, a PhaseGraphPanel for the first two independent variables (if there were more than one independent variables), a MultiGraphPanel, and a WorkspacePanel.

### 2.1.4 GraphPanel

The GraphPanel package contained a number of different classes defining the different panels that were displayed within a TabbedGraphPanel.

#### GraphPanel

The GraphPanel was a short abstract class from which each of the different kinds of tabs inherited. Rather than using proper inheritance structures, ComponentGraphPanel, PhaseGraphPanel, and MultiGraphPanel were virtually identical to each other, with large amounts of code copied verbatim. These child-classes were responsible for the toolbar above the graph and popup menus accessible by right-clicking.

#### ComponentGraphPanel

ComponentGraphPanel contained graphs of one dependent variable versus time.

#### PhaseGraphPanel

PhaseGraphPanel contained graphs of one dependent variable against another. It was largely identical to ComponentGraphPanel, with added functionality for plotting orbits and finding equilibria.

#### MultiGraphPanel

MultiGraphPanels contained graphs of multiple dependent variables against time.

#### WorkspacePanel

WorkspacePanels represented the advanced tab. Strangely, despite inheriting from GraphPanel, WorkspacePanel did not in fact include a graph. Rather it included curve information, including tables of points. Thus, most of the functions required of a GraphPanel were overridden and left empty.

### 2.1.5 PlotPanel

The PlotPanel subpackage exemplified the user interface's problems. PlotPanels were responsible for drawing graphs, displaying them within the interface, calculating window dimensions based on graphs, and calculating and displaying direction fields. All of this was handled by a single class, depending on the type of plot desired. The files in this package were generally poorly written and documented.

### PlotBox2D

PlotBox2D represented all of the ”background” parts of a plot, including the graph itself, labels, tick marks, and grid lines. It was responsible for drawing itself and contained modal information, plot ranges, pixel dimensions and a host of other functions. It was also responsible for panning and zooming.

### Plot2D

Plot2D inherited from PlotBox2D, and represented plots with only one dependent variable. It also maintained curve and direction field information and was responsible for drawing curves. It was used by ComponentGraphPanel and PhaseGraphPanel.

### MultiPlot2D

This class was similar to Plot2D, but was instead responsible for plots with multiple dependent variables (i.e., MultiGraphPanels).

# Chapter 3

# Design Improvements

Considering the problems and limitations of the original architecture, we created the following classes and packages to replace the old User Interface.

## 3.1 Control (Package)

The aim of the new Control package was to redirect input from the UI into more abstract, backend commands that could easily dictate program flow from a central source. This alleviates the need to route data between the various user interface classes.

### 3.1.1 Main (Class)

Main starts the program. Right now it simply begins the GUI, but it should be refactored with GUI to load the solvers, load the libraries, etc.

### 3.1.2 Listeners (Package)

Listeners notify the various classes of relevant user events, allowing them to transfer program flow to Control. Currently, this package contains many listeners from the UI, but some still need to be transferred over, and listeners with duplicate functionality need to be merged.

### 3.1.3 Data Representation (Package)

We have aggregated a significant amount of scattered and non-encapsulated data in the User Interface into a number of useful data structures, located

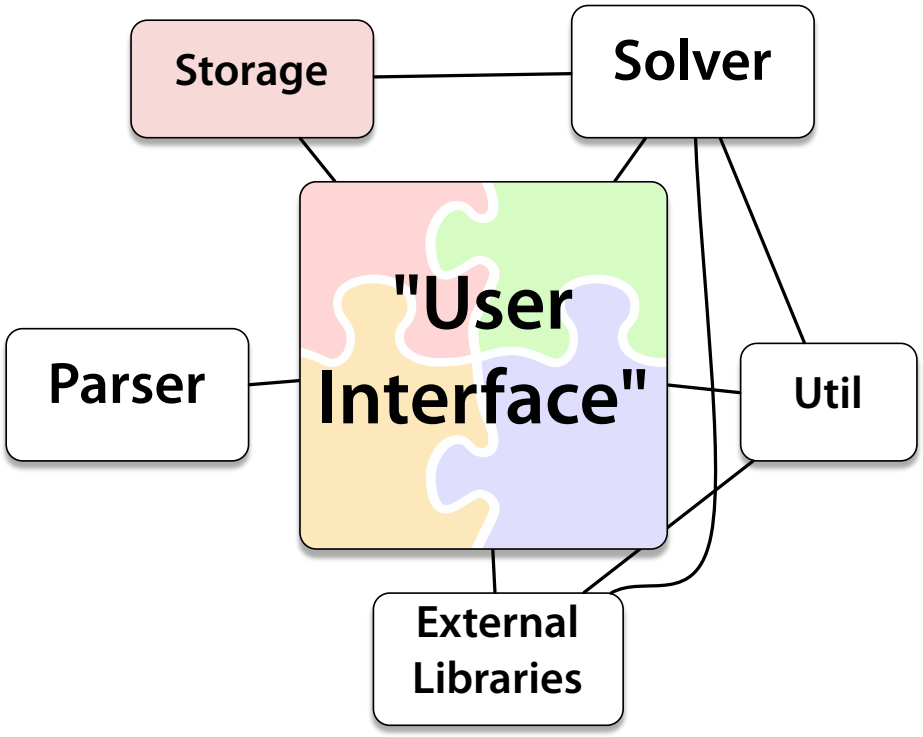


Figure 3.1: Diagram of the main components in the original architecture of *ODE Toolkit*, color-coded to indicate portions that were separated in the new design (Figure 3.2). See Appendix B.1 for summaries of packages in the original architecture.

in this package. Most important of these are the data structures used to encapsulate plot information and functionality.

**PlotState (Package)**

PlotState is a package containing the classes that store the internal representations of each plot panel. It contains the abstract class BaseState and its two children, SinglePlotState and MultiPlotState.

BaseState
: This is an abstract class that represents a generic graph. It contains a title, two Axes, a vector of Curves, a direction field equation, a

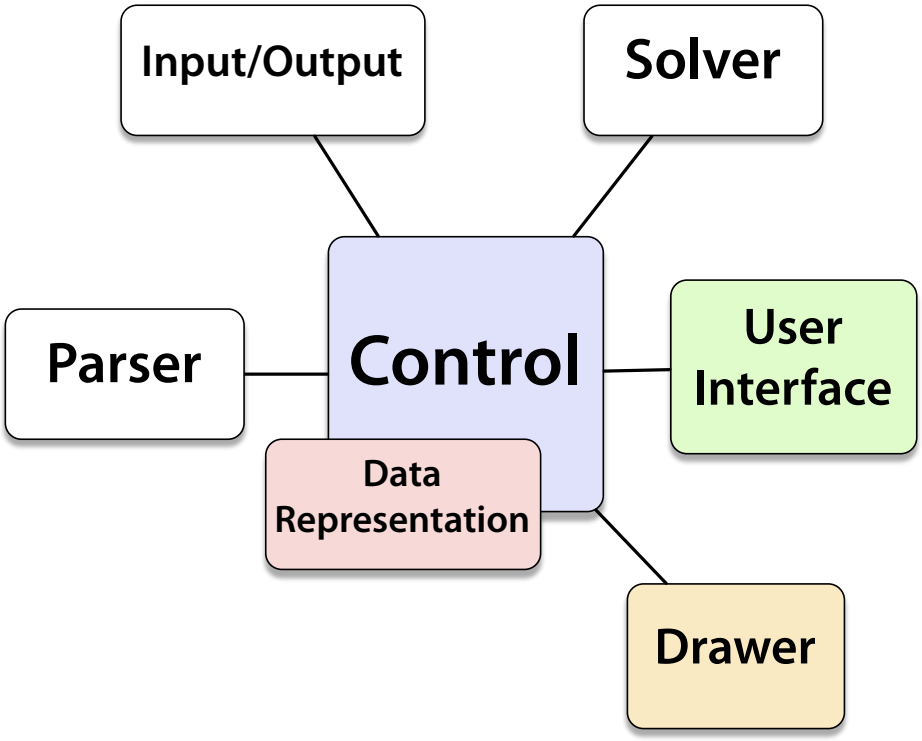


Figure 3.2: Diagram of the main components in the redesigned architecture of *ODE Toolkit*, color-coded to indicate portions that were separated from the old design (Figure 3.1). See Appendix B.2 for summaries of the new architecture's components.

curve color, and other important information. It also contains functions common to all types of PlotStates, such as changing Axis ranges. This completely separates a great deal of functionality and information from the user interface.

SinglePlotState

This stores the internal representation of a plot with a single dependent variable, and includes functionality for direction fields. This is the most common type of PlotState used, as it is used by all ComponentGraphPanels and PhaseGraphPanels through their PlotPanels.

MultiPlotState

This is used to represent plots with multiple dependent variables. As

different kinds of plots must be drawn differently, each PlotState has a DrawMySolutions function (defined as an abstract function in BaseState) which wraps a call to the appropriate function in Drawer (see below).

#### Axis (Class)

Axis contains an upper and lower bound, a label, and a number of computational functions. It has the information necessary to convert between coordinates in pixels and plot states if it is given its width in pixels as displayed on the screen. This class is also responsible for calculating placement of tick marks and gridlines.

By isolating the calculation of axes from the user interface, we were able to simplify much of the drawing as well. Changing the scale of the axis from linear to logarithmic is now all taken care of internally and invisibly to other components. It also includes some safeguards for resizing.

#### ODEVar (Class)

ODEVar is a simple class used to represent a variable, containing the variable's name and its position in the variable ordering being used by the program. That is, if points are stored in the form $(t, x, y)$ then the ODEVar for $t$ has position 0, the one for $x$ has position 1, and the one for $y$ has position 2. It was created because previously the variable's name and position were being passed separately, which led to messy, bug-prone, and non-intuitive code.

#### ODEVarVector (Class)

ODEVarVector extends the class Vector⟨ODEVar⟩ with a number of commonly used functions (such as providing a string with all variable names) included.

## 3.2 User Interface (Package)

### 3.2.1 TabbedOutputPanel (Class)

TabbedGraphPanel was renamed TabbedOutputPanel for consistency (see OutputPanel, below). In addition, we did some internal cleanup and refac-

tored the code to use classes such as ODEVar and to properly instantiate new OutputPanels.

### 3.2.2 OutputPanel (Class)

To rectify the unusual situation where WorkspacePanels (now renamed DataPanels for clarity) were treated as GraphPanels and then forced to override inapplicable functions, we created a new abstract OutputPanel class, which we made the parent of both the DataPanel and GraphPanel classes.

### 3.2.3 GraphPanel (Class)

GraphPanel was originally a small class, while its three children, ComponentGraphPanel, MultiGraphPanel and PhaseGraphPanel, were quite large. We have since removed all functionality common to these three child classes (which was mostly copied over verbatim) and moved it into GraphPanel. We also moved some formerly independent classes, such as PopupListener, into GraphPanel as internal classes. This left the GraphPanel parent class containing the bulk of the code, with its children containing only the code necessary to specify the slight variations.

### 3.2.4 PlotPanel (Class)

PlotBox2D, Plot2D, and MultiPlot2D have all been broken up and their components renamed. The parts of these files dealing with the internal representation of the graph's state were moved into the PlotState classes. The functions responsible for actually drawing the curves, plot background, direction fields, and so forth were moved into Drawer, where they are called by PlotState. The UI components, listeners, etc. were moved into the PlotPanel class, each of which contains a PlotState. Each GraphPanel contains a PlotPanel.

## 3.3 Drawer (Package)

Drawer is now a separate, highly compartmentalized class contained within its own package. Graph-specific elements, such as curves and direction fields, are drawn separately from generic elements common to every plot, such as titles, labels, and tick marks. All of Drawer's public functions take as input a Graphics object with which to paint. Currently, only PlotStates

call Drawer, through functions dedicated to drawing specific components of the type of PlotState given. The Drawer package also contains a class for generating PostScript output.

# Chapter 4

# Testing and Results

## 4.1 Testing

We decided to use manual testing for *ODE Toolkit* for two main reasons: first, since we were refactoring rather than creating new functionality, figuring out the exact intended output of each original function did not seem particularly productive; second, the changes we were making required testing the graphical output of the program, making automated testing difficult. Given the goals of our project, we felt that automated testing was unnecessary.

Thus, we determined a set of features for *ODE Toolkit*, which we then used as a basis for testing. We tested all features on multiple ODE systems of varying complexity. Please see Appendix C for a full testing procedure.

## 4.2 Results

The new version of *ODE Toolkit* accomplishes all tasks described in the Testing section, on Windows, Mac, and Linux systems. Saving and loading now works, unlike in the old version, as does printing. Logarithmic scale axes now work as expected: the user can enable logarithmic scaling on either the horizontal or vertical axis, or both. Panning now implements offscreen buffering, resulting in a much faster and smoother process.

The user interface has some new features as well. More features have been added to the toolbar, such as printing, exporting, panning and zooming. (See Figure 1.1 for a screenshot of the user interface).

In addition, the new *ODE Toolkit* performs all these actions considerably more quickly. Drawing more plot lines slowed *ODE Toolkit* to a crawl,

and Direction Field options took a significant amount of time to recompute slopes and repaint after each change. The result of our refactoring was an *ODE Toolkit* that handled all the above functions immediately, making the program much more responsive and less frustrating to use for students. Figure 4.1 lists some of the more significant benchmark times for *ODE Toolkit* performed on a 2008 Macbook Pro running Mac OS X 10.5. These tests are not rigorous, they simply demonstrate the speed issues present in the alpha 0.9 version. Typically, Windows and Linux performed better than Macs for Alpha 0.9.

| Test | Alpha 0.9 (s) | Alpha 1.0 (s) |
|---:|---:|---:|
| Resize window (at any time) | 4 | $< 1$ |
| Plot 10th solution curve | 7-9 | $< 1$ |
| Turn on Direction Field (with 10 orbit lines visible) | 14 | $< 1$ |
| Change length of direction field arrows | 27 | $< 1$ |
| Switch tabs with direction field activated | 6-16 | $< 1$ |
| Autoscale graph to fit in viewing window | 10 | $< 1$ |

Figure 4.1: Selected benchmark comparison in Mac OS X. The tests were run on a 2008 Macbook Pro, using the ODE $x' = y, y' = -5\sin(x) - y$.

# Chapter 5

# Addition Deliverables

## 5.1 Documentation

We believe that by thoroughly documenting our code and architecture that we will dramatically enhance what future teams will be able to accomplish. By providing a clear, concise explanation of each component's roles and responsibility in one centralized location, we hope to minimize the amount of time necessary for code familiarization, and ensure that future programming teams can quickly find pertinent sections of code, thus dramatically improving their productivity.

Our documentation has numerous levels. All functions are commented at a level appropriate for their complexity. All code files have a header clearly describing their purpose and dependencies. Every package has a file describing in detail what that component does and how it relates to other components, as well as a diagram or document detailing the purpose and requirements of its internal files and subcomponents. The repository contains a useful history log describing each major revision and the current direction of the project. The documentation has been compiled using Javadoc into an HTML linked guide and placed in the repository and on the *ODE Toolkit* website.

## 5.2 Developer Tutorials

We wrote a short tutorial for programmers that describes how to begin working on the project within the Eclipse integrated development environment, main architectural components, and how these components fit

together. It is included in the repository as a PDF file with its LaTeX source, and is also included here as Appendix D.

## 5.3 Example Library

We have compiled a small set of examples in the form of *ODE Toolkit* save files (.ode) containing common and informative systems of ODEs. The set of examples is included in the Library of *ODE Toolkit*. See Figure 5.1 for an example.

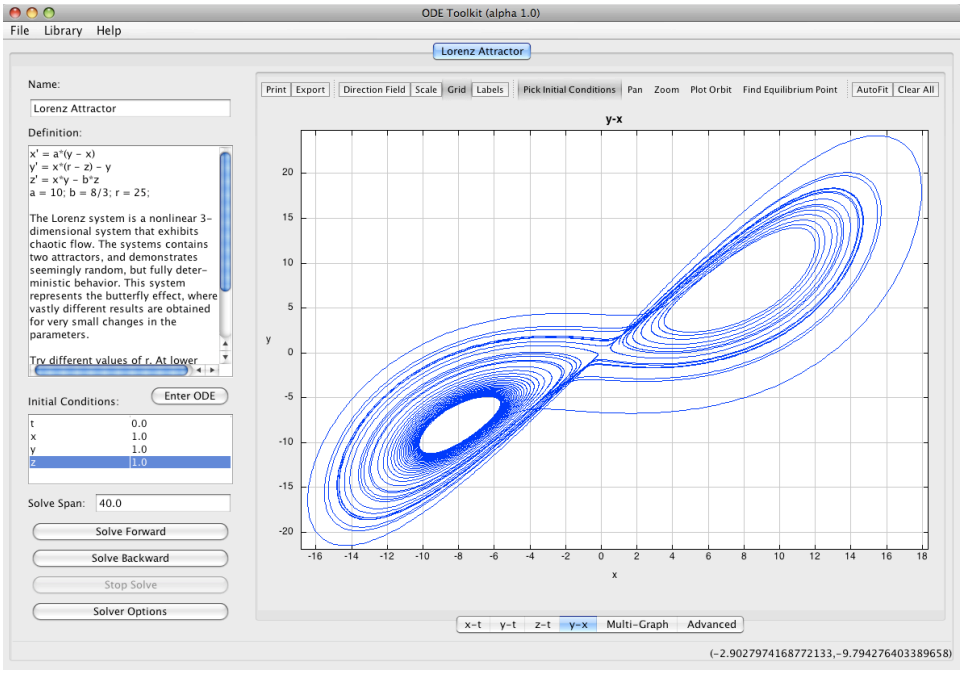


Figure 5.1: Sample ODE file: The Lorenz Attractor

## 5.4 Open Source License

One of the requests from CODEE was to make *ODE Toolkit* open source. There were a few requirements given to us to take into account when selecting a license. *ODE Toolkit* should be free to run, for any purpose. The source code should be made available, and open to be adapted as desired. The software should be free to distribute. Developers should also be free

to release improvements of the code. *ODE Toolkit* cannot be sold, none of the code can be patented, and the code may not be used for commercial software. Any parts of the code may be used in another program, provided that it gives credit to the authors and is distributed under the same license.

Given these terms, we have chosen version 3 of the GNU General Public License, as it meets all conditions on how the code can be used or adapted and is a commonly accepted and well-known licensing agreement.

We included a commented copyright notice in each Java file and included a license file in the repository containing the GNU GPL version 3 copyright agreement. The header comment is shown in Appendix E.

## 5.5 Trac Wiki

We have compiled a list of known bugs and suggested enhancements using the Trac ticket system, which will be available to future developers of the project. Trac allows developers to be able to choose tasks according to their interests and ambitiousness, and provides a centralized place for a team to manage the project's direction.

Listed below are some of the most prominent tickets:

- Bug: If the user attempts to solve while a popup window is displayed, the UI crashes.
- Bug: Printing does not seem to work under Linux.
- Bug: The UI does not automatically repaint after some popup commands.
- Improvement: Allow the user to plot analytical solutions to compare to numerical solutions.
- Improvement: Draw the grid lines on the offscreen plot rather than on the screen. This would allow the user to see the grid move while panning.
- Improvement: Reroute program flow to go through Control rather than ODEWorkspace
- Improvement: Improve the online help system.
- Improvement: Allow the parser to accept simple higher order differential equations.

# Chapter 6

# Conclusions and Future Work

In order to facilitate future development of the *ODE Toolkit* project, we have proposed a new internal design and made significant progress toward its full implementation. We have provided documentation and tutorials to ensure the continuation of this work, and also contributed additional materials providing immediate benefits to the project.

This section details necessary future work.

## 6.1 Completing Implementation of Control

There is still much to be done in cleaning up the flow of the program. We would ultimately like for all UI events to call functions in Control, which would then behave appropriately. Furthermore, the functions that have been moved to Control have not been refactored to eliminate redundancies.

## 6.2 Creating an I/O Package

The code that implements saving and loading files needs to be cleaned up. Currently the program uses code generated by Castor XML, which interfaces between Java and XML. However, the files generated for this purpose clutter both the Util and Util.XML packages. They should be separated and placed into a new package. There are other components that also seem like they should be placed in a dedicated File Operations package, such as those for exporting a plot as a GIF- or JPEG-encoded image.

## 6.3 Teacher Utilities

We would like to extend *ODE Toolkit*'s teacher support functionality in several ways. To begin, it would be useful if educators could annotate graphs and had an area available for a short problem statement or description. We also envision the example library supporting pre-implemented 'steps' that students can follow. Additionally, allowing educators to set up the Web Start program to load an ODE upon launching would greatly improve ease of use for students, although a tool to automate this process for educators would probably be needed as well.

# Appendix A

# Project Management

Despite our best efforts to keep a time buffer in the second semester, we ended up working all the way up to project deadlines, and even had to push them back at times. However, we suggest that this was due to initial confusion, overly optimistic goals, and an overall concern for the long-term future of *ODE Toolkit* beyond our own clinic project.

## A.1 First Semester

Figure A.1 shows our work schedule for the first semester. In hindsight, we should have been able to accomplish more during the first semester. This would have made our second semester work much more reasonable. Several factors contributed to the situation, described below.

### A.1.1 Initial Setbacks

The project suffered from a slow start. We faced a significant difficulty setting up the Eclipse platform to work with the *ODE Toolkit* repository and becoming familiar with the repository structure. We were also delayed by the late receipt of our clinic machines, on top of the time and effort that went into setting them up. An additional setback occurred when we found that the most current version of the *ODE Toolkit* program, version Alpha 1.0 (not to be confused with our final release), contained numerous bugs that rendered it barely functional. As a result, we had to search through the previous revisions to find the latest working version, Alpha 0.9, which took additional time. Many of these impediments can be avoided in the future with proper documentation and proper use of version control. Our

developer tutorials should allow future developers to avoid such difficulties.

### A.1.2 Initial Optimism

In our initial scheduling, we underestimated the complexity of the architecture and, subsequently, the amount of time required to become sufficiently familiar with it to attempt refactoring. We originally allocated about two weeks of debugging for familiarization, leaving documentation to be completed after our refactoring effort. However, at the end of the debugging phase, we felt that debugging alone would not give us a comprehensive understanding. Documenting the current architecture seemed to be the most efficient path toward understanding it, with the added benefit that significant amounts of the documentation would still be relevant to the finished project. Thus, we spent about three weeks completing package and file overviews; function-level comments were added after refactoring. In order to compensate for the unexpected delay, we considered abandoning the port of the *ODE Architect* example library, as any future workers on the project should be capable of doing this in our stead. In addition, we reduced our time commitment to refactoring the architecture from six weeks to five. As it turned out, we needed far more time than five or six weeks, especially since we were unable to finish writing formal specifications for our proposed design before the winter break.

## A.2 Second Semester

Figure A.2 shows our work schedule for the second semester. Even in the second semester, our overly optimistic outlook persisted for several weeks. This resulted in a significantly back-loaded schedule.

### A.2.1 Initial Design and Refactoring Attempt

First, we spent a few weeks planning our new design. We proposed a set of new classes and sketched out a few sequence diagrams to test them out. After that, we began refactoring. We started first with simple modifications to the package organization, then began to pull apart the plot-relevant data from the user interface components to create a PlotState data structure. We also attempted to separate the drawing and control functionality from the user interface components.

### A.2.2 Design Phase

Unfortunately, it soon became apparent that we could not simply divide up the old classes into new ones; we ran into a number of issues with our proposed design, and needed to ensure that our new architecture made sense for every possible use of the program. Thus, we decided to stop refactoring and plan out our new design in much more exact detail, creating sequence diagrams for all sufficiently complex functions, and ensuring that all classes had the necessary information to perform their duties straightforwardly. This took a significant amount of time to complete.

### A.2.3 Refactoring

By the time we finished designing, we only had a few weeks to actually implement our changes. On top of that, we decided it would be best to undo our changes from the initial refactoring attempt and start clean due to the large changes in design we had since made. However, with the design more properly fleshed out, the refactoring process itself became much more straightforward. With prudent re-prioritization of our other academic coursework, we were able to accomplish a great deal of work in this short amount of time.

Gantt Chart (Task) (Clinic)

| | Name | Start Date | Effort | Duration | Resource Short Names |
|---|---|---|---|---|---|
| 1 | Debug/Documentation | 2008-10-11 | 100 | 6w 4d | E, R, S, A |
| 2 | Prepare Phase 1 Presentation | 2008-10-17 | 20 | 1w 2d | A, E, S, R |
| 3 | Testing | 2008-10-24 | 40 | 2w | E, S, R, A |
| 4 | Prepare Phase 2 Presenation | 2008-11-19 | 40 | 6d | E, S, R, A |
| 5 | Mid-year Report Draft | 2008-11-26 | 20 | 1w 2d | A, E, S, R |
| 6 | Mid-year Report Final | 2008-12-05 | 20 | 1w | A, E, S, R |




Figure A.1: Tasks accomplished in the first semester

Gantt Chart (Task) (Clinic)

| | Name | Start Date | Effort | Duration | Resource Short Names |
|---|---|---|---|---|---|
| 1 | Design Phase 1 | 2009-01-21 | 60 | 2w | A, E, S, R |
| 2 | Refactoring Phase 1 | 2009-01-27 | 90 | 4w 5d | E, R, A, S |
| 3 | Open Source License | 2009-02-22 | 12 | 1w | A |
| 4 | Phase 3 Presentation | 2009-03-11 | 60 | 1w 1d | A, E, S, R |
| 5 | Design Phase 2 | 2009-03-02 | 120 | 4w 4d | E, S, R |
| 6 | Projects Day Poster | 2009-04-03 | 40 | 1w 4d | E, S, R, A |
| 7 | Refactoring Phase 2 | 2009-04-09 | 160 | 2w 3d | A, E, S, R |
| 8 | Testing | 2009-04-27 | 10 | 4d | A |
| 9 | Fixes/Enhancements | 2009-04-27 | 30 | 4d | A, E, R |
| 10 | Projects Day Presentation | 2009-05-02 | 80 | 3d | E, S, R, A |
| 11 | Final Report | 2009-04-28 | 40 | 1w 4d | A, E, S, R |
| 12 | Tutorials | 2009-05-04 | 15 | 5d | S |




Figure A.2: Tasks accomplished in the second semester

# Appendix B

# Documentation

## B.1 Package Summaries (Old Architecture)

Unless otherwise specified, these packages were retained in the final version.

### B.1.1 Parser

The Parser is mostly automatically generated by JavaCC, and is used to translate the equations entered by the user into a form usable by the program. It also has hand-coded definitions of the various functions supported, such as addition, subtraction, floor, sine, etc.

### B.1.2 Solver

Like the parser, the solver package is already well-defined and modular. It consists of a collection of different numerical solvers with each sharing a common interface, but not all written in Java—non-Java solvers do not work on non-Windows platforms.

### B.1.3 Storage (Replaced by DataRepresentation)

The Storage package defines a data hierarchy for working with the plotted data points. It consists of three classes: Workspaces, ODEs, and Curves. At the top level, Workspaces represent a single tab open in the program. Workspaces contain a collection of ODEs, each representing one equation that has been entered in that Workspace. In turn, ODEs contain multiple Curves. At the bottom level, data points are stored as arrays of doubles in

Curves; each time the user hits the ”Solve” button again, a new Curve is created to contain the points generated by the solution.

### B.1.4 Util

Util provides a variety of functionality, primarily file saving and loading, which should probably be handled in a separate package. This makes up more than half of the Util folder, and is entirely automatically generated by Castor, a software utility for storing Java objects in XML. There are also files for navigating a file system, which would fit well in a file I/O package. In addition, there are files for finding equilibrium solutions that are not currently used. Finally, Util contains a utility to extract non-Java solver files from the distributed JAR, as well as a few other miscellaneous files.

### B.1.5 External Source (Removed by refactoring)

There are a number of small packages from external sources, typically open-source code that is used to handle a number of mundane tasks such as displaying the opening splash screen. Some of these, such as IntHashtable.java in the Acme package, are obsolete. Others would likely be better served as members of other packages.

## B.2 Package Summaries (New Architecture)

This section describes the architecture’s new packages.

### B.2.1 Control

Control currently consists of three files, plus the DataRepresentation sub-package:

**Main**
: Currently, the Main class is a wrapper that calls the GUI() constructor, ”turning on” the program’s user interface.

**CentralStorage**
: CentralStorage is used for storing the image used in panning, and may ultimately be used to store ODEs as well as any other objects that must be universally available.

**Listeners**

The primary purpose of Listeners is to ensure that all program flow is handled in one place, thus reducing the ”spaghetti code” feel and allowing greater reusability. All listeners to any program component should eventually be placed within this file (which may at some point be expanded to a full package). Currently, listener functions are grouped into internal classes according to the component which uses them. The subclass naming convention is:

“Parent class name” + L.

Thus, ToolkitMenuBar’s listener functions are in the internal class ToolkitMenuBarL.

Ultimately, every component should have its own unique listener function, even if it does the same thing as another component. This is to prevent unnecessary interdependencies, to maintain consistency (having this function wrap another function is, however, fine), and to ensure that two user interface components that do the same thing can be easily decoupled if one needs to be changed.

Functions should be given all information that they will need, such as mouse click data or a pointer to the user interface.

Although listeners will still be added by the individual components, each listener will now refer to a function in this file. Keeping all the listener definitions in one place will make it easier to extend and maintain, and will keep program flow more clearly defined.

Further information on listeners is available in the top-level documentation of the repository as GeneralListenersInfo.txt.

**DataRepresentation**

DataRepresentation is a collection of data structures used throughout the program.

**Axis**

This class is the internal representation of an axis. It knows its range, is capable of computing where tick marks should be placed, can convert plot coordinates into pixel coordinates given a pixel length and vice versa, handles scaling (and includes the various necessary safeguards) and handles conversion to and from logarithmic scale.

**Workspace**
The Workspace class contains the representation of a single file tab. This is the backend that ODEWorkspace interacts with. It contains a vector of all ODEs that have been entered in that tab. The Workspace also takes new ODEs from the user interface (i.e., the ODEWorkspace), sends them to the parser, and creates new ODE objects to represent them.

**ODE**
The Workspace holds a collection of ODEs, each representing one ODE that has been entered in that workspace. This contains the representation of the actual ODE itself. Primarily, however, it is responsible for dealing with the Curves associated with any given ODE.

**Curve**
Each Curve represents a single solution. Curves contain a collection of points generated by a solution given a particular set of initial parameters. Currently the Curve's points can be changed by a setter, which is poor encapsulation and should be changed.

**ODEVar**
This simple class is designed to contain the internal representation of a variable, namely its name and the index reserved for it within the points list. For instance, if points are stored $(t, x, y)$ then $t$ has position 0, $x$ 1 and $y$ 2.

**ODEVarVector**
Extends Vector<ODEVar>, providing some additional useful functionality. Always use ODEVarVector instead of Vector<ODEVar>.

**PlotPoint**
Not currently widely used, PlotPoint extends Point2D.Double with a single boolean added, which represents whether it is connected with a line to other points.

**PointComparer**
PointComparer compares the x-coordinates of two plotpoints. Implements Comparator<PlotPoint>. Possible target for refactoring.

**Tick**
A Tick includes a tick-mark's value (as a String), its axis-location, and whether its label uses scientific notation.

**PlotStates**
This package contains classes for data structures representing plots. Details are described in Chapter 3.

### B.2.2 Drawer

Drawer provides classes for the creation of graphical representations of plot states. It can draw Graphics objects for display on the screen or for exporting PostScript output. The Drawer creates the graphical representation of the plot states. There are overloaded functions that take specific types of plot states. PlotStates call drawPlotBackground first, which draws the plot's frame (titles, labels, tick marks, grid). Then they call drawCurve for each of their curves, passing in a separate graphics object from the screen one. This offscreen buffer is usually larger than is visible, to make panning more efficient.

### B.2.3 UI

UI contains the program's user interface. Some functions handling program flow control remain here, but we have changed many aspects as well. The following classes and packages are new to the code:

**OutputPanels**

Provides an abstraction of tabbed panels for displaying different forms of output, including graphs and data. Subpackages contain various different kinds of OutputPanels, plus PlotPanel, which is not an OutputPanel but is embedded in GraphPanels.

**OutputPanel**
An OutputPanel is a generic abstract class that contains some basic functions. It is the parent of DataPanel and GraphPanel.

**TabbedOutputPanel**
A TabbedOutputPanel is essentially a collection of the various OutputPanels used by a graph.

**OutputPanels.DataPanel**
The DataPanel subpackage handles the "Advanced" display tab for a given ODE tab. The most important class is also called DataPanel, and consists of a split pane: on the left is a Java swing tree view of the

stored ODEs and their curves, and on the right is a CurveInspector panel, which displays the specific data points in the selected curve. (Each separate solve action creates a new curve, and changing the ODE makes a new ODE in the hierarchy. These are not to be confused with new ODE tabs, made with the `File->New` command.)

**OutputPanels.GraphPanels**

The GraphPanels subpackage is responsible for showing graphs and handling the associated UI elements. It also contains PlotPanel, which though not itself a GraphPanel is embedded within GraphPanels.

**GraphPanel**

GraphPanel is an abstract class used to hold a two-dimensional graph and its associated UI components. GraphPanels all contain a name and a vector of curves. They have methods for printing and exporting PostScript output, receiving data, and setting and maintaining plot variables. GraphPanels have a toolbar which includes buttons for displaying directional fields, printing, and clearing, and also include a great deal of functionality common to all GraphPanels.

**ComponentGraphPanel**

ComponentGraphPanels are GraphPanels that represent a graph of two of an ODE's variables. However, they are currently used only to plot single dependent variables against time. They are created and contained in the ODE's TabbedOutputPanel, which currently creates one for every variable used in the ODEWorkspace.

**PhaseGraphPanel**

PhaseGraphPanels are very similar to ComponentGraphPanels, except they have additional "Plot Orbits" and "Find Equilibria" features. Currently TabbedOutputPanel creates only one, for the first two dependent variables, and there is no way of creating more.

**MultiGraphPanel**

MultiGraphPanels draw plots of multiple variables against a single independent variable. They are currently used to plot all variables against time ($t$). It would be nice to be able to select which variables are shown.

**PlotPanel**

PlotPanels are the panels on which the actual curves are plotted,

and include gridlines, axes, tick marks, borders, and labels. Each PlotPanel wraps a PlotState, so its parent GraphPanel interacts with its PlotState through the PlotPanel. In addition, they contain a number of mouse listeners that would be difficult to move into Control.Listeners.

## B.3 Sequence Diagrams

Because of the size of the project and the number of packages involved in most tasks, we decided that it would be helpful to create a series of sequence diagrams detailing how program control and information was passed between different classes for various different common actions. We have reproduced all of our sequence diagrams below.

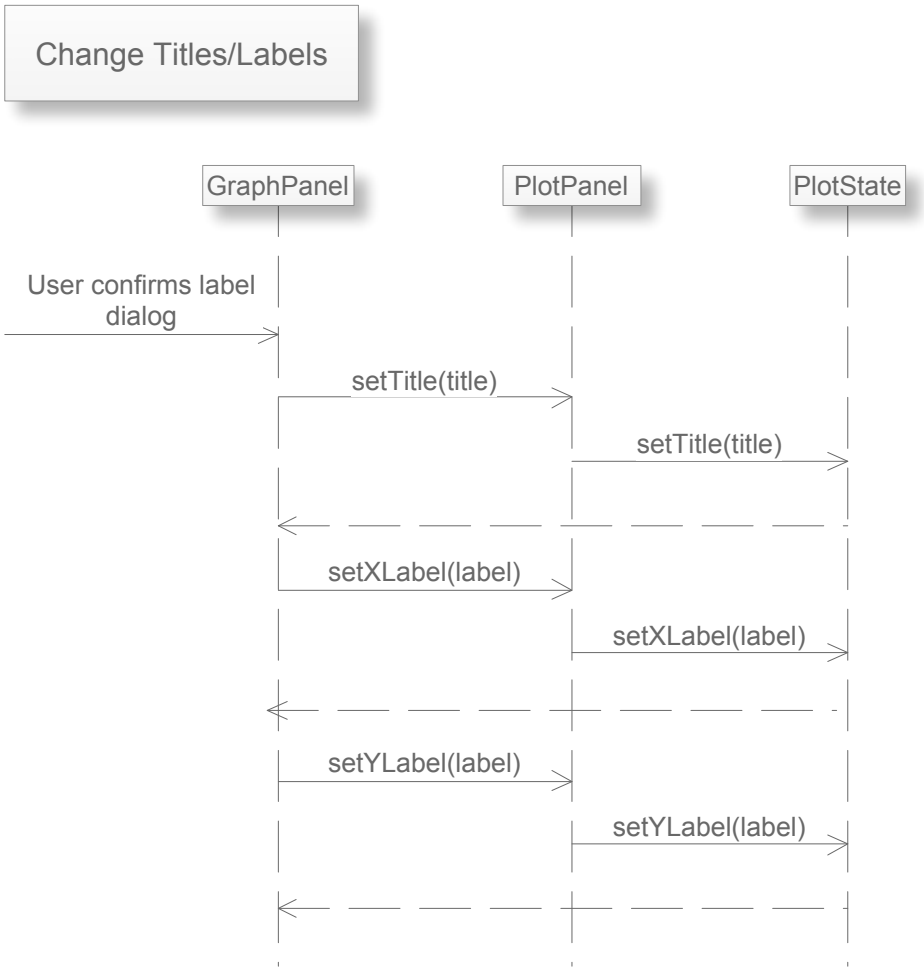


Figure B.1: Changing titles and labels.

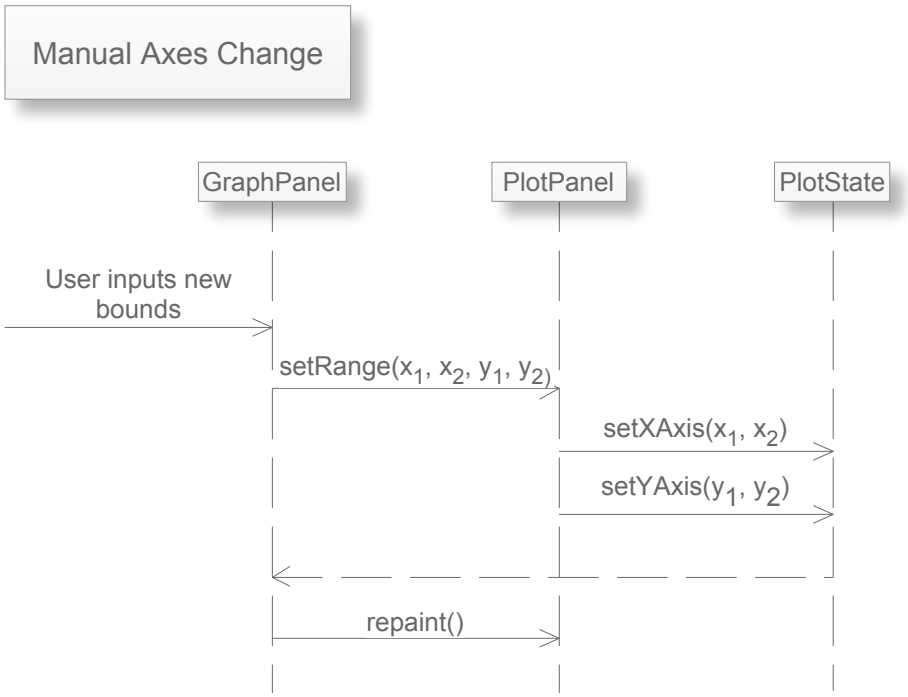


Figure B.2: Changing axes' ranges manually.

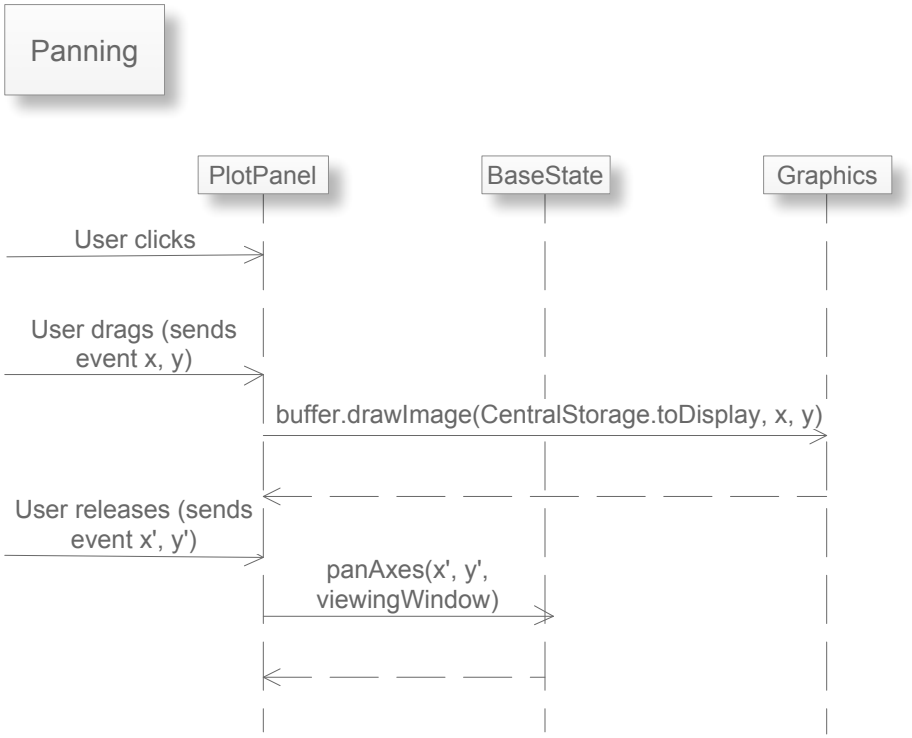


Figure B.3: Panning. Upon the first click, the PlotPanel stores the event location as the origin. As the user drags, the PlotPanel pulls the panning buffer image in CentralStorage, translates it appropriately, and draws it to the screen buffer. Upon mouse release, the plot is shifted to complete the pan.

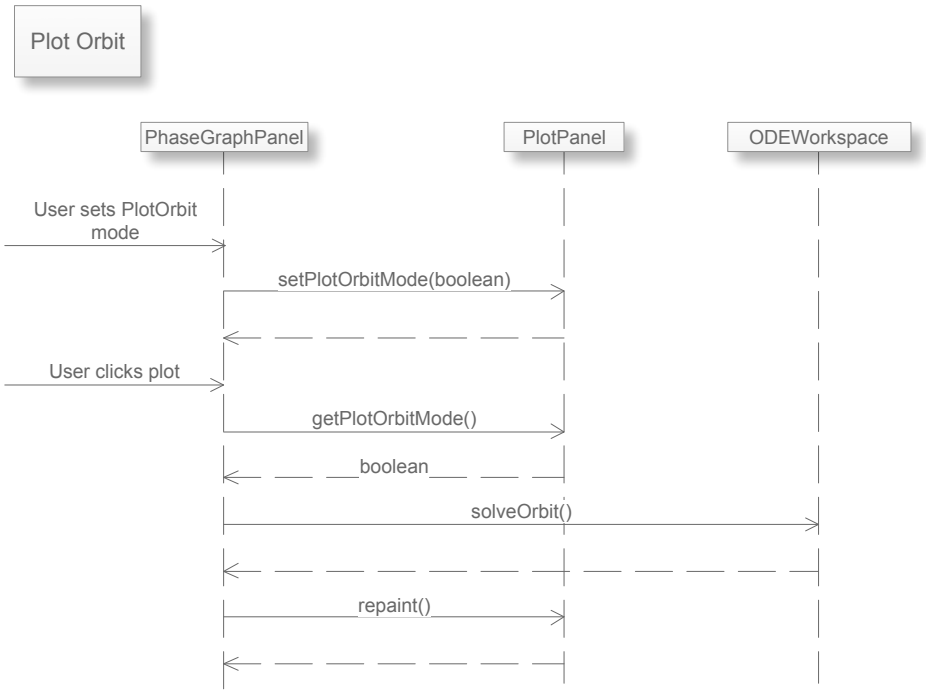


Figure B.4: Plotting orbit lines. When the user clicks, the PlotPanel checks to see if the program is in Plot Orbit mode. If it is, then it plots the orbit accordingly.

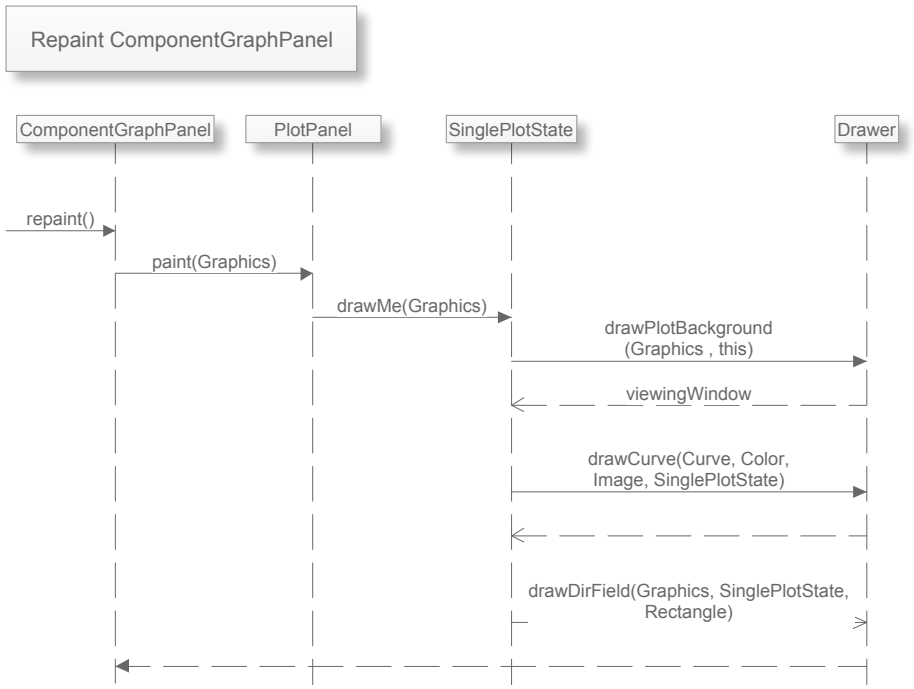


Figure B.5: Calling `Repaint()` on ComponentGraphPanel.

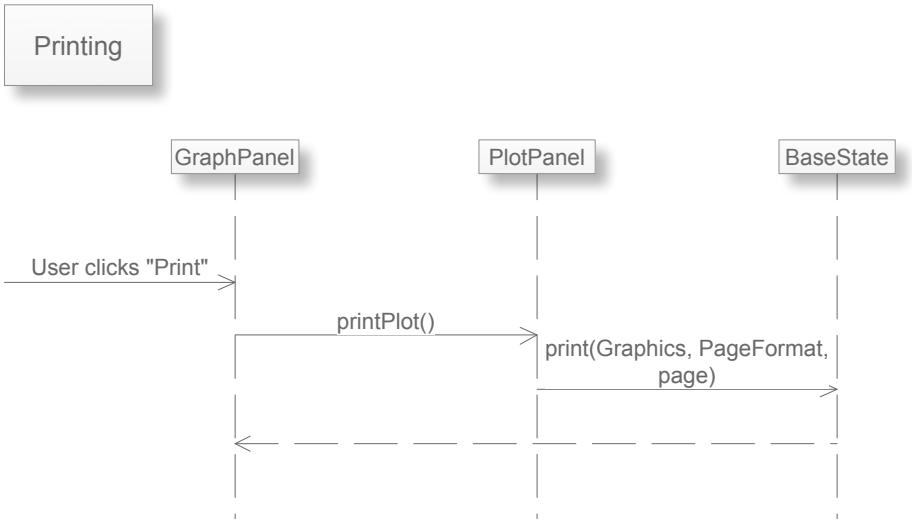


Figure B.6: Printing

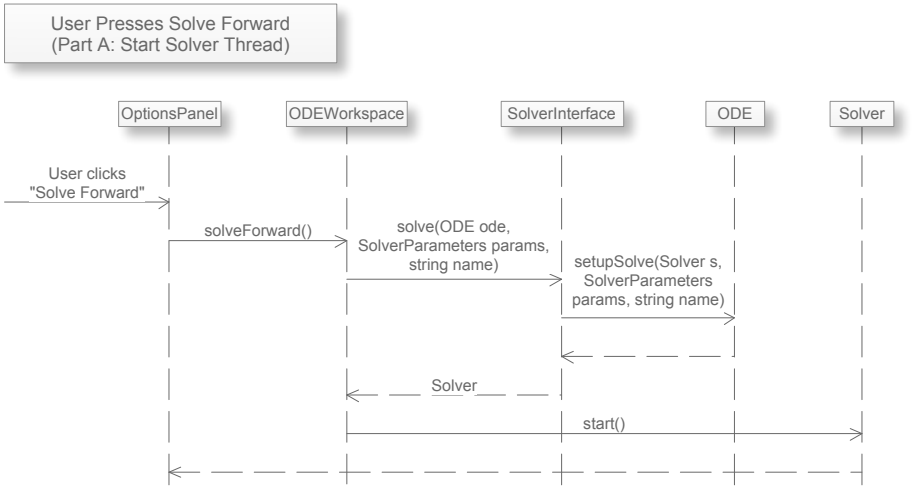


Figure B.7: Solving forward. In this first step, the solver thread is started.

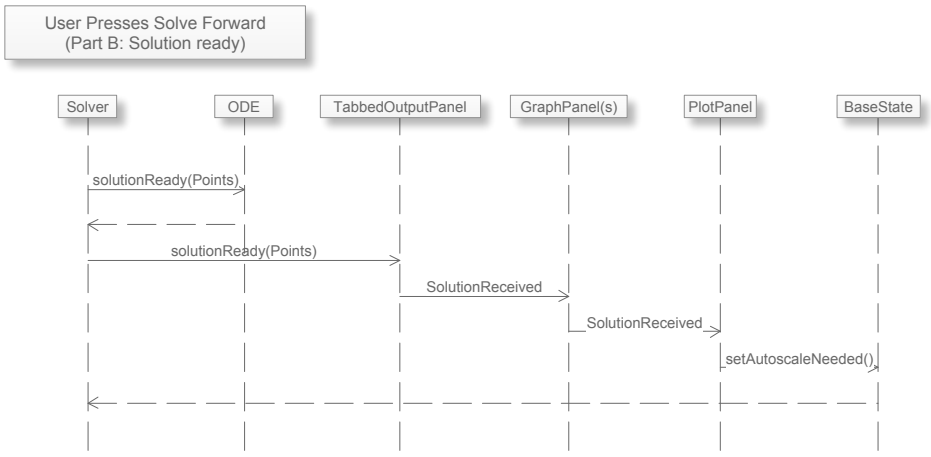


Figure B.8: Solving forward. When the solver thread finishes, it notifies the central ODEs to add the new points and the TabbedOutputPanel to update its GraphPanels.

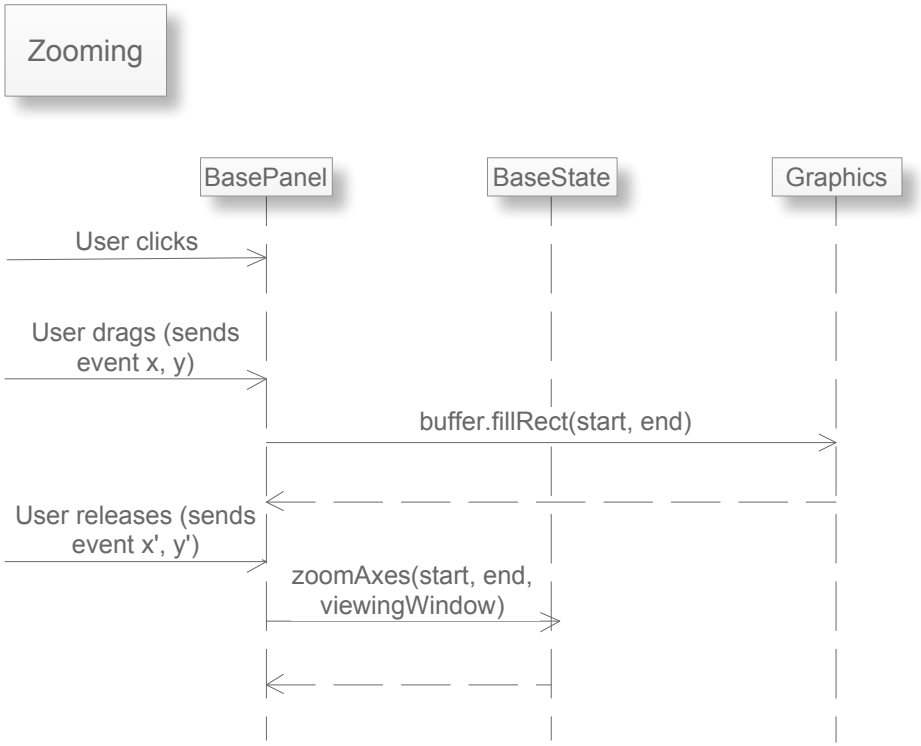


Figure B.9: Zooming. Upon the first click, the PlotPanel stores the event location as the origin. When dragging, the PlotPanel draws a zoom box on the screen buffer. Upon mouse release, the plot ranges are updated to complete the zoom.

## B.4 Class Diagrams

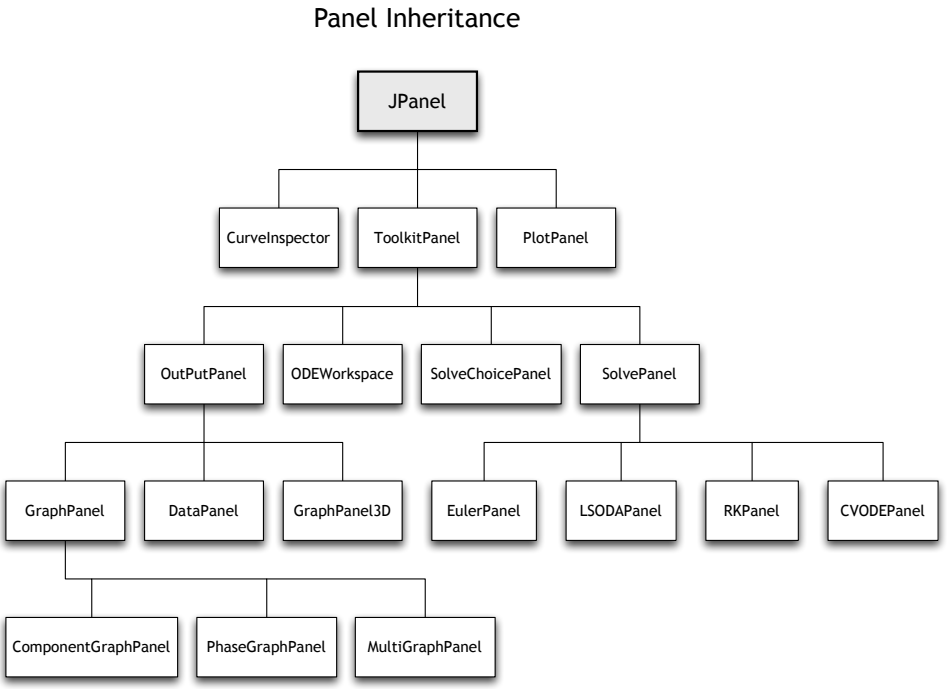


Figure B.10: The user interface includes a large number of different types of panels; this diagram illustrates their inheritance hierarchy.

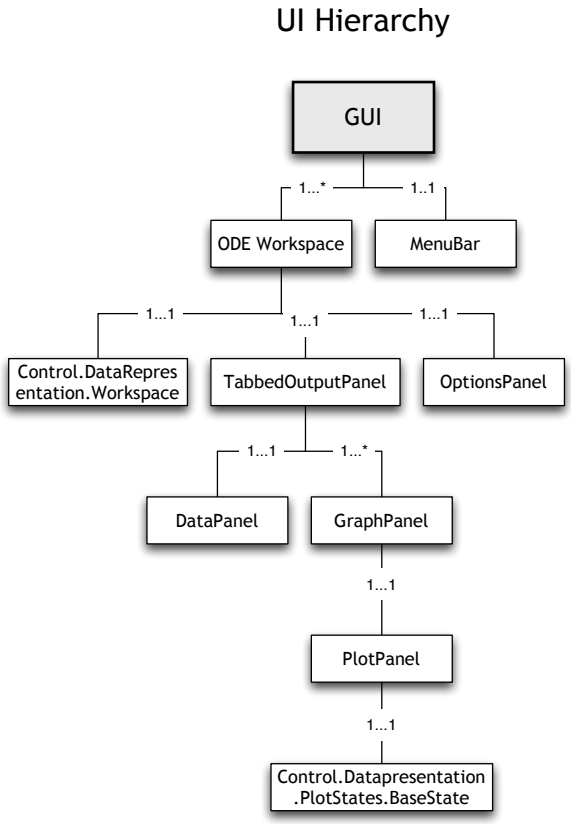


Figure B.11: The user interface consists of a hierarchy of components; this diagram highlights the UI's containment hierarchy—that is, which classes contain instances of other classes as members.

# Appendix C

# Testing Procedure

The following are a series of manual tests for *ODE Toolkit*. This is designed to be a robust testing procedure.

1. Enter the equation $x' = x$. A blank graph with grid lines on, title "x-t", axis labels "x" (horizontal) and "t" (vertical), grid lines on, and the "Grid" and "Pick Initial Conditions" buttons selected, should appear. The window should be -1.1 to 1.1 for both $x$ and $t$. Try turning grid lines on and off.

2. Without changing initial conditions, hit solve forward. A horizontal line should appear stretching from 0 to 10. The $t$-axis should rescale from 0 to slightly more than 10. The $x$ -axis should stay the same.

3. Without changing initial conditions, hit solve backward. The line should now go from -10 to 10. The t-axis should also expand accordingly.

4. Click on the graph somewhere random above the line. Initial conditions should change appropriately. Hit solve forward. An exponential curve starting where you clicked should appear. Plot a few more curves, forward and backward, and make sure autoscaling works reasonably.

5. Attempt to click as close to (0, 0) as possible. Check to make sure the initial conditions look reasonable.

6. Right-click on the graph. Make sure that "Pick Initial Conditions" mode is selected here too. Now select "Pan". Make sure that the

"Pan" button in the toolbar also selects. Play around with the different modes and make sure that the toolbar and the popupmenu's statuses always match, but when you're done come back to "Pan".

7. Click and drag on the graph. The grid lines and tick marks should disappear and the curves should pan according to where you move your mouse. Let go, and the gridlines and tick marks should reappear, and the tick labels should update.
8. Click "Autofit". It should return you to where the graph was before you started panning.
9. Switch to "Zoom" mode.
10. Click a point on the graph, and drag down, to the left or the right. A blue box should follow your mouse cursor, anchored to where you originally clicked. When you release, it should zoom in to the area you selected. Zoom in very close around 0, and make sure scientific notation works.
11. Click again, and this time drag up and to the right. A small guide box should appear within the blue box. Make the blue box about three times as wide and long as the guide box, and release. The graph should zoom out, with the range expanded by about a factor of 3. Zoom out very far, and make sure scientific notation works.
12. Try manually scaling the graph via the "Change Window Ranges" option in the scale menu. Hit Autofit again to return to where you were.
13. Right click and click Direction Field. A direction field should appear. Do this again, to turn it off.
14. Go to Direction Field on the toolbar. Try turning DirFields on and off from here. Make sure the popup menu's dir field checkbox follows the state of the toolbar menu's dir field checkbox.
15. With direction fields on, open the direction field menu and try to change colors, density, length, and arrowheads.
16. With dir fields still on, start playing around with log axes. Make sure the dir field changes appropriately, and make sure the checkboxes stay in sync. Make sure scientific notation works. Make sure neither axis includes anything $\leq 0$.

17. While still testing the above, try zooming. Make sure tick marks, both horizontal and vertical, are reasonable, especially making sure to avoid label overlap. Make sure one cannot cause any kind of overflow or underflow errors, and that one cannot zoom too far out or too far in. There should always be at least three orders of magnitude between the start of the log axis and the end.

18. Finally, set the vertical log axis on and the horizontal one off, and make sure the exponential curves now appear linear. They should all have the same slope. Turn log axes off.

19. Check all of the above (except for direction fields, which should be off) in multi-graph panel.

20. Look at the curves in the advanced panel. Do the points look reasonable? Try deleting a curve, and make sure they actually delete.

21. Enter the system of equations

$$\begin{aligned} x' &= y \\ y' &= -x \end{aligned}$$

    Plot a few graphs. They should be sine curves.

22. Do all of the above tests in $x$-$t$ (with the exception of direction fields, which should be off). Make sure that your old exponential graphs are still there. Make sure that $y$-$t$ also works, although extensive testing should not be as necessary.

23. Check the $x$-$y$ plot. It should appear like an ellipse (in point of fact it's theoretically a circle, but it won't appear that way unless the viewing window is square).

24. Try all of the above tests. Direction fields should work here. Solving forward or backward should both produce ellipses centered at 0.

25. Try to find an equilibrium point. There should be one at (0, 0).

26. Try plotting orbits. This should produce ellipses that pass through the point you clicked.

27. Do the above tests in multi-plot. Make sure that your old exponential curves are still there.

28. Enter the system of equations

$$\begin{aligned} x' &= z \\ z' &= -5 * \sin(x) - z \end{aligned}$$

    Graph a few solutions, making sure your old x solutions remain on the *x*-*t* graph.

29. Go to *z*-*x*, and plot a few solutions forward, and plot a couple of orbits.

30. Go to multi-graph, and play around here as well. Make sure that your old $x$ and $y$ plots are still here.

31. Try printing and exporting.

32. Try saving and loading. When you load a saved file, both input and output panes should match what was in the saved file.

# Appendix D

# Developer Tutorial

## D.1 Getting Started with Eclipse

Eclipse has been the primary editor used on this project. Here's how to get an Eclipse project up to speed quickly. To install Eclipse add-ons, from `Help` in the menubar click on `Software Updates`. Then select `Add Site` and give it the appropriate URL. The URL for the packages we use is listed below. Then select the packages you want and install.

**Subclipse**

Subclipse is an Eclipse tool which connects an Eclipse workspace to a subversion repository. To install, you can follow the instructions above, using a URL:

`http://subclipse.tigris.org/update_1.4.x`
or alternatively following the instructions at

`http://subclipse.tigris.org/install.html`
Once you have Subclipse installed, you can begin working by selecting

`File -> Import -> Checkout Projects from SVN`.
The URL should be for the specific branch you want (as the entire repository is very large). The repository URL for the trunk is (as of May 2009):

`vetinari.math.hmc.edu/svn/odetoolkit/branches/Alpha9`

**Copyright Wizard**

Copyright Wizard is an Eclipse tool that streamlines the process of adding copyright headers to the source files. To install, use a URL of

`http://www.wdev91.com/update/`. Then the license and header files can be edited in `General->Copyright` under `File->Preferences`. When all the settings are specified, the license headers can be applied by selecting `Project->Apply copyright...`. In addition, the `license.txt` is generated and included in the project.

**Fatjar**

Fatjar is a tool to combine various Jar files into a single Jar, and has been used to combine *ODE Toolkit* with Castor and other useful Jar files. To install, use a URL of `http://kurucz-grafika.de/fatjar`. To export, select `File->Export` and select the Fat Jar Exporter. Specify the Main-Class to be Control.Main, leave One-Jar unchecked, and uncheck `select Manifest file`. When selecting files to include in the Fatjar file, some of the Jar files can be left out to decrease file size. Any `j3d` files, as well as `vecmath`, can be left out if 3D plotting is unimplemented. `swing` must be included for the GUI to work, and `xerces` and `castor-logging` are (currently) required for saving and loading to function properly.

## D.2 Main Components

**Parsing**

The ParserInterface is the interface for compiling strings into functions. Most of the code is created by JavaCC and JJTree and the grammar is stored in `grammar.jj`. Terminal functions that the parser understands must be wrapped, and added to the list of known functions in the ParserInterface.

**Solving**

Solvers are threads, and solving is a side-effect. When an ODE (an already parsed equation) is being solved, it gives the solver thread a reference to a curve, and the points are added as the thread chugs along. When all the points have been added and the solving is done, the solver thread then notifies the SolutionReadyListeners, and is done. The points are already in the ODE at this point, which in turn means they are already in every PlotState that refers to them.

**PlotState**

This is the plot. It has the data points, the axes, the options, and is the model collected into a single place.

**Drawing**
Drawer is a static class with no variables, only constants. It is where drawing takes place. Its public methods take in a graphics object and the data they are supposed to draw.

**Saving/Loading**
Saving and loading is done in XML. Currently, we are using something called Castor XML, which automatically generates code to turn java objects into XML. This currently resides in Util, and needs cleanup.

**UI** There are many different components to the user interface, some of which are panels for no good reason.

**ODEWorkspace**
is the current interface between the UI and the other modules.

**Input**
optionsPanel is the class where differential equations and their initial conditions are entered, as well as where the Solve buttons reside and where you can access solver options. Solving stuff gets passed to workspace panel to solve, and parsing stuff gets passed up to the workspace which call parse. This should get rerouted to activate Control instead.

Popup Menus and Buttons are input from the user manipulating the output or its format. These mostly reside in the GraphPanel (buttons for changing mouse mode, options, etc.) but the functionality which is dependent on interacting with the graph (panning, zooming) is carried out in PlotPanel. Again, stuff that affects the internal state should be changed to call a Control function, rather than grabbing it through the PlotPanel.

**Output**
Graph Panels: These contain both Plot panels, as a way of holding the output, and various forms of input to manipulate the plot. Most of the common functionality between different types has been refactored into a parent class.

Data Panel: This is an output panel that displays solved points in a data table, called the 'advanced' tab in the UI. It does not contain a Plot Panel.

Plot Panel: This is a dumb panel. Its primary purpose is to hold the plot. It also can pan, zoom, and figure how mouse clicks interact with the PlotState.

Unfortunately, this is being used for a lot of sending information to PlotState, which should NOT go through here and instead should go through Control.

**Control**

How to think about the flow of the program:

Right now, it mostly works like this: user input is either handled in place, passed down to the PlotState through a series of wrappers, or passed up to the ODEWorkspace class to ship out to other modules. How it should work: User input activates control function, that either changes the state directly, or activates another module directly.

## D.3 Miscellaneous

### D.3.1 Java Web Start - Digitally Signing the Jar

In order for Java Web Start to run a program that can access system resources, the program must be digitally signed and trusted by the user. To sign a Jar file requires keytool and jarsigner. (There's probably a package which combines these into Eclipse but we didn't look for one).

keytool creates keys to allow you to cryptographically sign Jar files. (Someone should probably look into publishing the public key to a trusted site.) keytool should be built-in on Macs and Unix-based machines. To create a key:

```
keytool -genkey -alias CODEE
```

where `-genkey` says that you want to create a key, and `-alias` asks for the name that you will call this key— nobody sees this, so it doesn't really matter, but we'll call it CODEE. It will then prompt you for two passwords. The first password protects all the keys stored in your keystore, and the second is the one you need to encrypt or sign your message. If you have created keys before it will prompt you for the key store password. Once generated, the key will be stored in a file in your root directory called '.keystore'. If you want to save this in another place, use the `-keystore URLTOSAVETO` option, and your keys will live at URLTOSAVETO. Creating a key only needs to be done once, but you need to know both the key password and the key store password.

Eventually, there should probably be a key in the repository specifically for this project.

`jarsigner` allows us to sign the Jar files. First compile a Jar normally (we'll pretend it's called ODETOOLKIT). Then run:

```
jarsigner ODETOOLKIT
```

It will prompt for your key store and key passwords, and then it's done. That's easy enough. Some more options are useful though. If you want to make the signed Jar file a new copy, then the `-signedjar FILENAME` option works:

```
jarsigner -signedjar ODEToolkitWebStart.jar ODETOOLKIT
```

Also if the keystore is not in the default place, you can specify it with `-keystore URLOFKEYSTORE`.

### D.3.2 Javadoc

Javadoc is a tool for generating API documentation in HTML format from comments in source code. It should be used consistently to document each class as well as all non-trivial functions (excluding, for example, getters and setters).

Javadoc syntax can be seen in most classes in the project, and is similar to block comment syntax (`/** ... */`). Note that these comments must be placed directly before the function or class declaration.

To add overview- and package-level documentation, a separate HTML file must be added in the directory of the package, titled package.html (or overview.html). See

```
http://java.sun.com/j2se/javadoc/
```

for details, and look at existing documentation for examples. For simplicity, please try to consolidate all package documentation into the Javadoc output.

Compiling Javadoc is simple with Eclipse: with the root directory of the project selected, select

```
Project -> Generate Javadoc.
```

A dialog will allow Javadoc to be generated for different levels of visibility; for this project, create documentation for members with private visibility. Selecting `Next >` allows the documentation title as well as the overview HTML file to be specified.

Eclipse will create the documentation in the root level of the project in a directory called `Doc`. To view at the documentation, start at `index.html`.

### D.3.3 Adding Solvers

A new solver must extend the class `Solver` and implement methods `run()` and `kill()`. Furthermore, all points, *including the initial conditions* must be reported through `notifyPointReadyListeners()`. To evaluate the deriva-

tive of the variables, use `parserInterface.evaluateODEs()`. More detailed instructions can be found in the code repository, in `toolkitNG/toolkitclient/Docs/AddingSolvers.txt`.

### D.3.4 Managing the *ODE Toolkit* Website

The *ODE Toolkit* website is currently hosted at `vetinari.math.hmc.edu`. In order to access the server, you will need a math department account; ask Professor Yong if you need one.

The server does not allow users to connect directly through the `ssh` command. Instead, `ssh` into one of the machines in the math computer lab (e.g. `ssh USERNAME@cain.math.hmc.edu`). This can be done through a command terminal or a graphical frontend for secure file transfer (e.g. Fugu for Mac OS X).

The actual website files are in `/home/bc/odetoolkit-website`. If adding a new version of the program, place it in the `downloads` directory and update the link in `download.html`. If adding a new version of the Web Start program, update the links on both `home.html` and `download.html`.

# Appendix E

# Licensing

The following appears as a header comment in all source files of *ODE Toolkit*.

```
This file is part of ODE Toolkit: a free application that for
solving systems of ordinary differential equations.

Copyright (C) 2002-2009 Eric Doi, Andres Perez, Richard
Mehlinger, Steven Ehrlich, Martin Hunt, George Tucker, Peter
Scherpelz, Aaron Becker, Eric Harley, Chris Moore

This program is free software: you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation, either version
3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be
useful, but WITHOUT ANY WARRANTY; without even the implied
warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.  See the GNU General Public License for more
details.

You should have received a copy of the GNU General Public
License along with this program.  If not, see
<http://www.gnu.org/licenses/>.
```