Building Large Software Systems in *Mathematica*®

Building large software systems in *Mathematica* should follow the general principles that apply to building any large software system. The details may be unique to *Mathematica* but many of the principles are quite general. In addition, there are some extra techniques for which *Mathematica* is particularly suitable. You should be aware of these and take advantage of them.

These principles are relevant for any development other than quick prototyping of tools for rapid exploration. Systems for which these principles are relevant include the following:

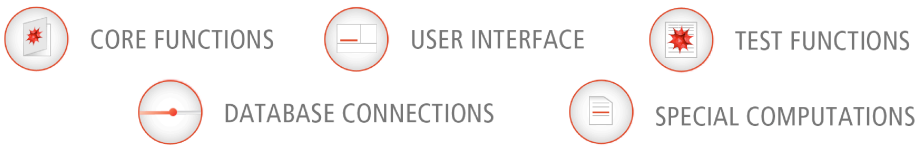
- a system to be used by other people, for example, in a commercial setting
- a system to be developed by more than one developer
- a system to be used over any length of time

Much of this document describes the development process for a system and how the system should be organized internally. End-users, of course, do not really care how something is developed, or how it is organized internally. Sometimes developers follow this philosophy as well: "if the user does not care—why should I?" But users do care about consequences that stem from good development practices. They care that a system has fewer bugs, performs better, has more features, and is delivered more quickly. They also care that they do not have to pay as much for it.

These principles have been derived in part from our experiences in building large *Mathematica* projects. A major example is *Mathematica* itself, much of which is implemented in the *Mathematica* language.

Divide the System into Components

You should divide your system into components; this is probably the most important principle to follow. If you have a satisfactory and natural division, then many benefits will follow without a lot of effort. There is nearly always a natural way to divide things up into different components. You should identify these components and build each as an independent sub-system.



Here are some reasons why building things as components is useful:

Components Can Be Developed Faster

Individual components are simpler to understand, have fewer goals, and require less implementation. Thus, there is less code to write to start with and less code to work with later. This is particularly useful for new developers starting on the project; they will have less to learn to become effective.

Components Can Be Developed Independently

If you have a team of developers, you can divide the components around the team and take advantage of any specific knowledge and expertise they have. You can make sure that a database component is given to someone who knows about databases, and ensure that those who do not know about differential equations are not exposed to code that requires knowledge about differential equations.

Components Can Be Tested Independently

Testing—as an integrated element of the development process—is a key part of successful software development. Working with components helps the testing process enormously. Some components, such as user interfaces, are very hard to test, but others are easier. Thus, testing is much easier if the system is broken down into components. In addition, testing individual components helps to quickly identify which component is causing an error when there is a test failure, and this speeds up resolving the problem.

Components Can Be Replaced

Development is often a continuous process. It is quite common for changes to occur, such as new requirements being produced or new technology becoming available. Working with components helps to switch over to a new way of doing something. For example, a new type of database becomes available. If all the connectivity is put into a database component, this will make it much easier to replace the component with a new one.

Components Can Be Used in Unforeseen Ways

A division into components gives much more flexibility. For example, if you have a special way to do interpolation, another application might want to use it. If this is in its own component, then it is easy to provide to the other application. Alternatively, for a badly segmented application you might be tempted simply to copy the implementation. This would be bad since you would end up with two copies of the same thing, requiring bug fixes or new features to be added in two places. Yet another problem would come if you provided your entire application simply to give the functionality from just one part. This would also be bad: there might be intellectual property, trade secrets, or licensing problems. The best way is just to provide the component that is wanted.

Another positive aspect to using components is that you could much more easily switch your system to a web delivery system using web*Mathematica*.

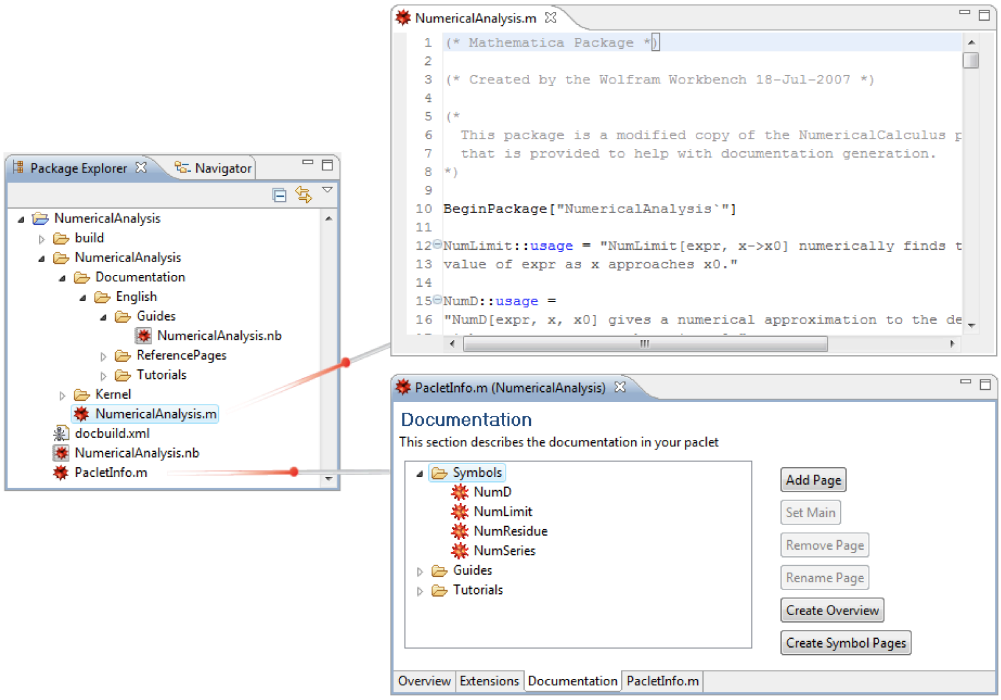
Think of the Architecture, Not the Code

This is closely related to the principle of dividing into components. It means that you should try to think about the large granularity of the system, in particular its components, rather than all the details of the code. If you have a good architecture for your project, then the code will follow quite easily. It is much harder to create a good architecture than it is to write code. Many people can write code, even without a lot of training or experience. In addition, the consequences of bad code are much easier to deal with than bad architecture.

One principle says that the last thing to do when building a system is to write any code. You should do everything you can to avoid actual coding. Instead you should focus on your architecture, making tests, and writing documentation. Too many developers take the attitude that they should "dive in and start to code furiously"; often they would do better to stop and think about what they are building.

Use *Mathematica* Code Packaging

Mathematica has a number of features for dividing code into packages and applications. It is important that you make full use of these. In many cases you should put all your code into a package, one that starts with `BeginPackage` and ends with `EndPackage`. Your packages should be bundled into an application: this is the way that your application will be delivered. An exception to this is the bundling mechanism for *Mathematica* Demonstrations.



It takes a little bit of time to set things up as packages and applications, but *Mathematica* provides tools such as Wolfram *Workbench* that are designed exactly for this purpose. They give many productivity features that greatly facilitate working with *Mathematica* for software development.

Keep Things Simple

This principle relates mostly to the actual implementation. It suggests that it is better to write small functions and to use simple syntax to express things. This can be particularly important with *Mathematica*, which has an extremely rich syntax. It is also a good idea to benefit from the extremely literate programming style that *Mathematica* supports.

Another thing to avoid is creating strange control structures, for example overloading `BeginPackage` and `EndPackage`. It is an interesting feature of *Mathematica* that you can do this, but such things are nearly always to be avoided. These are just going to be an impediment to anyone else who wants to look at the system; they will also limit the utility of your code.

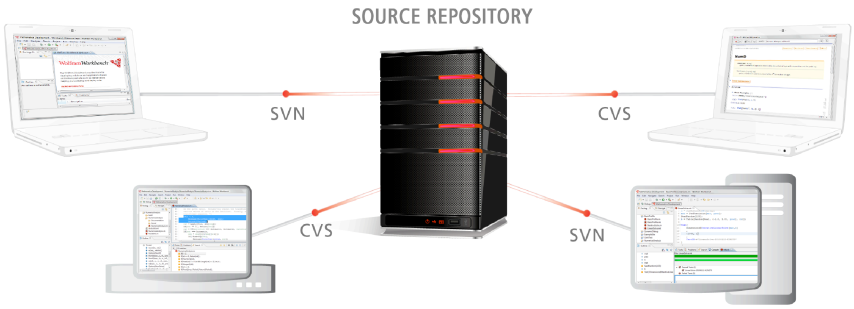
Use Source Control

Any system that you want to use and work on for more than one day should be in source control. This is a critical issue if there is more than one developer or if the project is developed over any lifetime longer than a few months.

Examples of source code control systems include CVS, SVN, and Rational ClearCase. They provide general features such as storing different versions of component files, recording a description of the reasons for a change to a file, tagging groups of files for a released version, comparing different versions of files, and merging changes. Source code control is essential for teams of developers, but is still extremely useful for a project developed by a single developer.

Wolfram *Workbench* contains a client for CVS, and is being updated to contain a client for SVN. It can be updated to

hold client support for most other source code control systems.

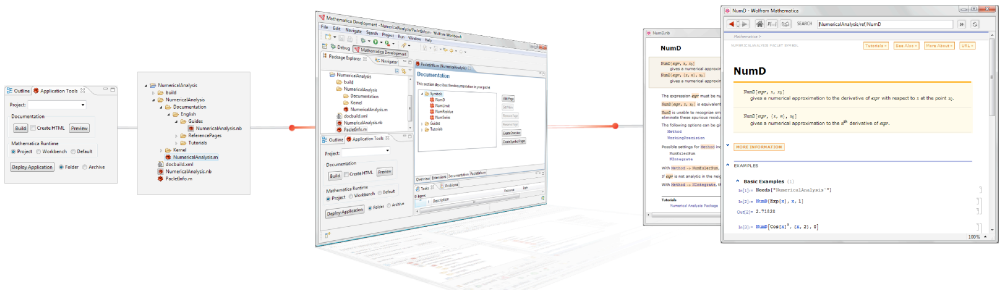


To use source code control you need to have access to a source code server or repository. These can be set up without too much effort on your own machines. Alternatively, there are a number of public systems. Note that these are only suitable for applications where it does not matter if the source code can be seen publicly. In addition, many organizations provide source control as part of their standard IT infrastructure.

In addition to source control, there are also build systems and configuration management tools. These work with your source control to build released versions of your application that you could give to end users.

Write Documentation

There are a number of different ways to document your application, depending on the audience.



End-User Documentation

Mathematica provides a documentation system that works with *Mathematica* applications and integrates with the *Mathematica* Documentation Center, and also can generate HTML documentation. Alternatively, your system might have its own documentation.

Good end-user documentation will help your system to seem more professional, and make it easier to use. But it can also help with the development process. It is often true that something difficult or awkward to document is not designed properly. Thus end-user documentation should be done as an integrated part of development.

Developer Documentation

This is documentation designed for developers, and it can take a number of different forms. You can provide comments in the code, and use names that describe the functionality. In addition, when you use source control you can describe the reasons for making changes as you commit changes. Other opportunities include storing documents in your projects. You can easily do this with *Mathematica* notebooks, and Wolfram *Workbench* provides a system for hyperlinking from source code into a notebook.

If you have written your code and then later change the way it works, for example, by using a different algorithm, you should definitely take the time to update the names that are used. It is very much preferred to use names that describe the actual implementation. Wolfram *Workbench* has tools available to help with renaming. You also should not be afraid to change the names of files that are used for implementation or even the name of the whole system.

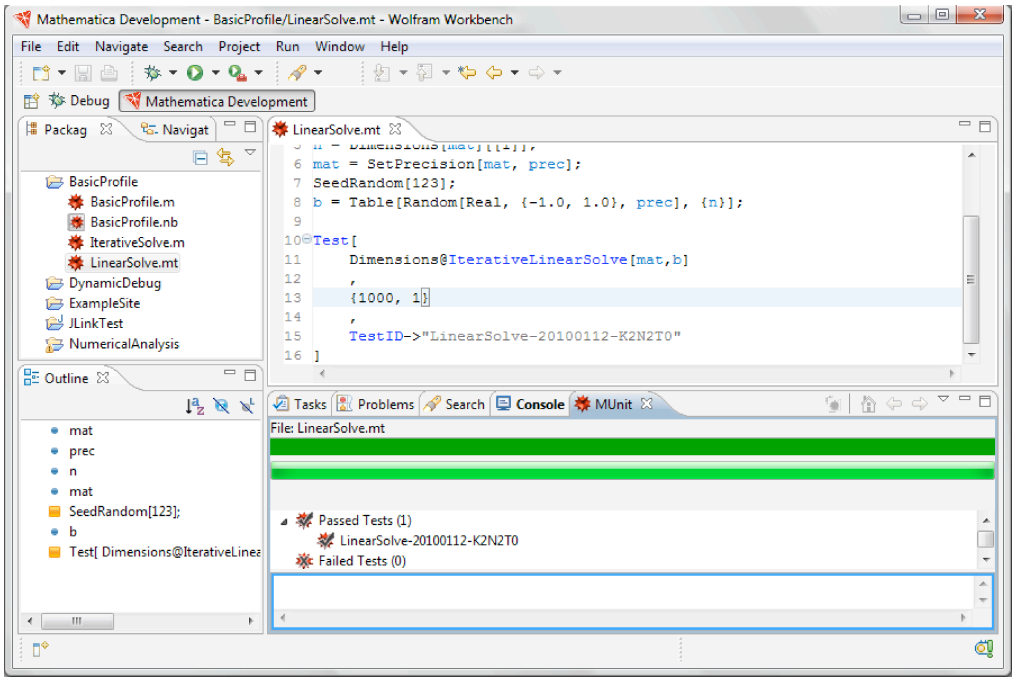
One naming rule is to avoid using `"new"` as a prefix. If you introduce a new way to do interpolation you should not call the function `"newInterpolation"`. The reason is that this will not be new forever and might end up as the oldest part of the system, but with a completely inappropriate name. Even worse, you might replace the functionality again, and then you cannot use `"new"` again.

Typically, at Wolfram Research we put a lot of effort into naming of our internal code, and we are quite willing to rename functions, variables, files, etc. when it is required.

Write and Use Unit Tests

Testing should be a key element of the modern development process. An older view of testing is that it was only done by a separate group of people whose purpose was to test the system. While a software quality department plays an essential role, testing should play an integrated role in development and should be done by developers as they produce code.

Mathematica code lends itself well to unit testing for a number of reasons, such as the ease with which data can be saved and restored, and the ease with which different *Mathematica* functions can be called. It also provides a very nice unit testing package called MUnit, which is often run through Wolfram *Workbench*.



You should develop tests as you work, making sure that any new feature is covered. If you find any bugs or weaknesses in your code, then you should immediately write tests when you fix them. You should run the tests frequently; for example, you should always run them before you commit to source control. You should also maintain the tests so that they always all pass with 100% success. To make good use of tests you should ensure that your system is broken into different components. Then you can have a different suite of tests for each component.

You will get a number of benefits from continuous testing.

Bug Fixing Is Easier

Running the tests frequently will let you catch bugs much more quickly. Generally, if you find a bug as soon as it is introduced then it is much faster to fix. A bug that you find six months after it is introduced is harder to fix. So running tests helps to boost your development productivity.

Refactoring Is Easier

Refactoring means changing the implementation of your application without really changing what it does. This might mean renaming, it might mean breaking the system up differently, or it might mean using a different algorithm. For all of these, if you have a good set of tests that all work after your changes, you will have more confidence that your changes are good.

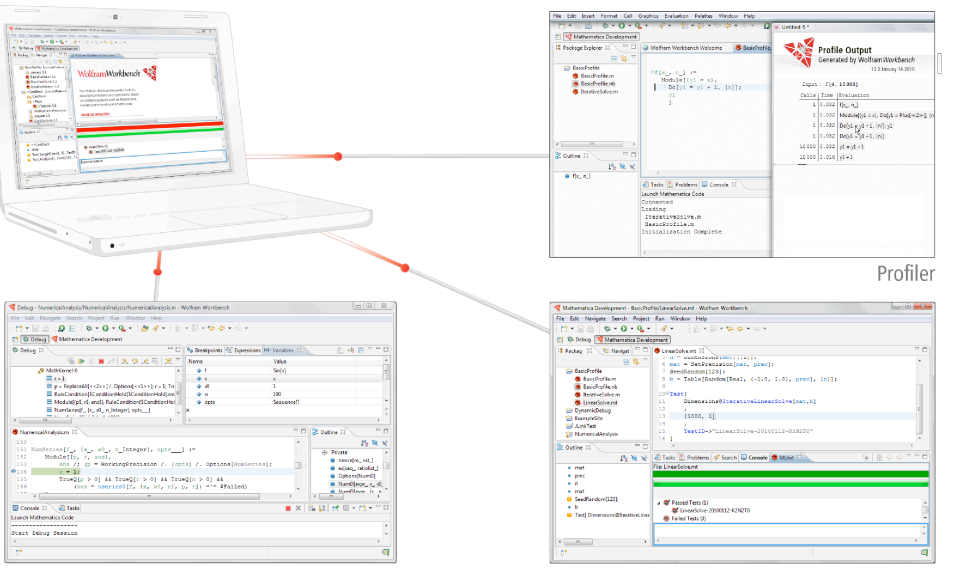
Updating to a New Version Is Easier

When you update to a new version of *Mathematica*, you can run your tests with the new version. This will give you confidence that you can upgrade to the new system.

Training for New Developers

When new developers join a project, working on improving and enhancing unit tests is often a good way for them to learn about the project.

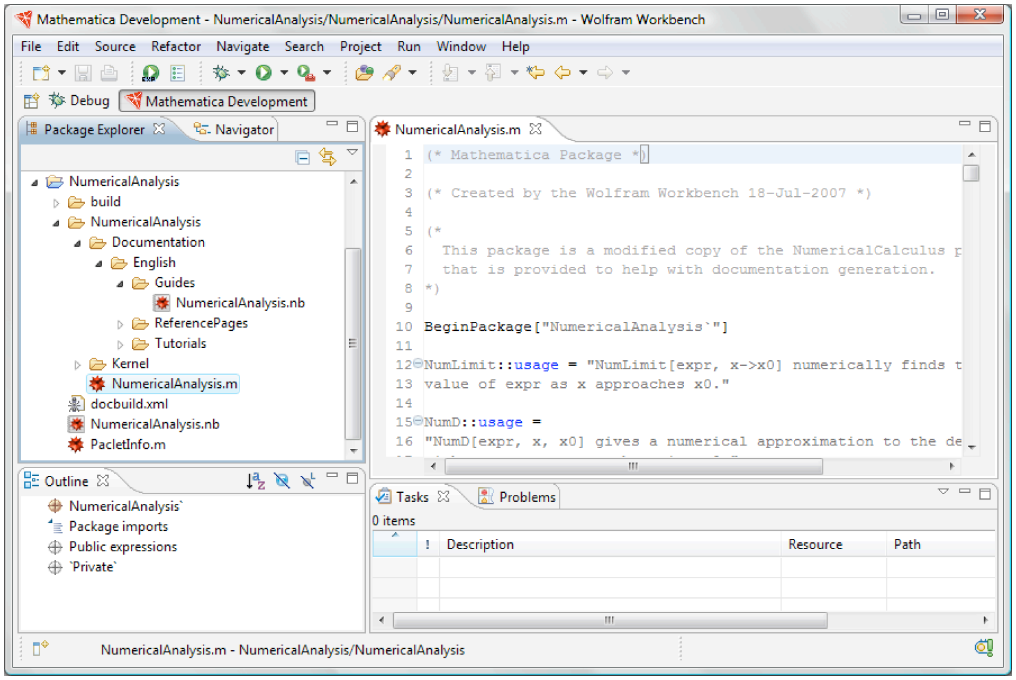
Use Wolfram *Workbench*



Wolfram *Workbench* is an integrated development environment for *Mathematica.* It is based on a widely used IDE platform called Eclipse, one that is used in many commercial environments. You can use *Workbench* directly or you can install the *Mathematica* tools into Eclipse.

Workbench provides many useful tools for building large applications. It contains a debugger, a profiler, and a unit tester. It also supports projects that contain different types of resources, such as code, documentation, Java classes, and notebook documents. This project view on your work fits nicely with breaking your system up into different components, which is one of the important ways you can boost development productivity.

Workbench also contains a special editor for *Mathematica* code and has a lot of knowledge about *Mathematica* packages and applications. This all helps you to develop and work in this way. It also contains good integration with CVS and can easily be extended to work with other source control systems.



It has a lot of support for developing in other languages, such as Java, Python, or C and C++. So if your system needs to integrate these with *Mathematica*, you can do so very readily.

One interesting feature of *Workbench* is that it gives you tools that help in working with parallel computation in *Mathematica*. You debug and profile parallel programs, helping you to add parallel programming to your system.

Here are some specific features of *Workbench* that help with large projects.

Project Organization

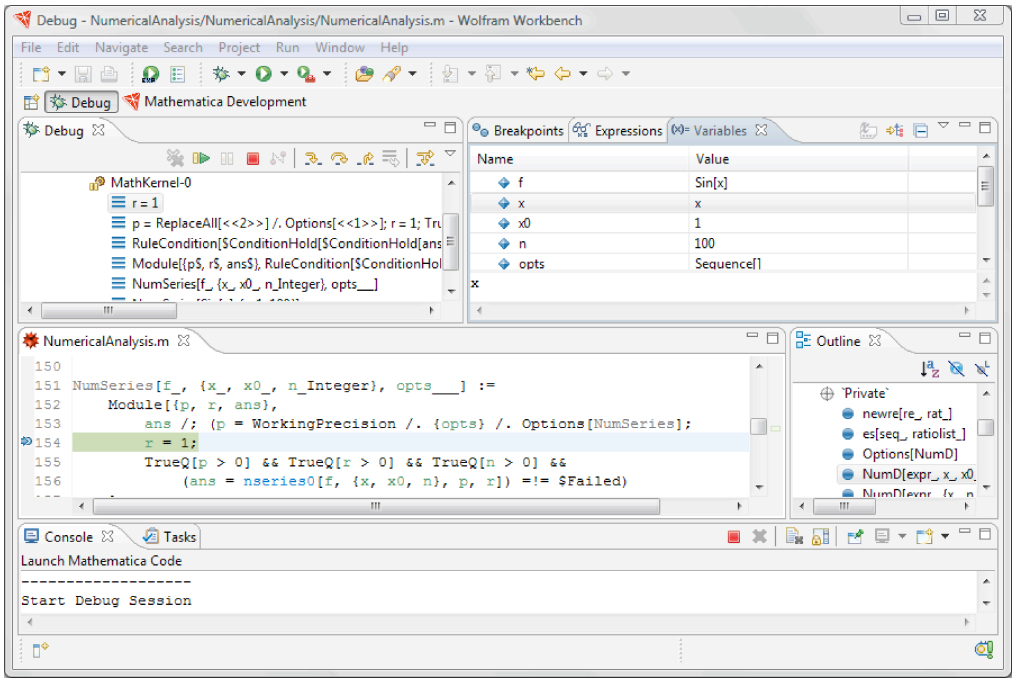
All your work in *Workbench* is collected into projects, each of which contains different types of resources, such as *Mathematica* code, documentation, Java classes, and notebook documents. Unlike some project-based systems, projects in *Workbench* are particularly simple and easy to maintain. *Workbench* has many special tools for working with projects, such as special editors for opening documents, specialized searching, and reports for different components. It also integrates tightly with the *Mathematica* notebook front end.

Launching and Running Code

Workbench provides a sophisticated interface for launching and running the *Mathematica* code from the project. It allows you to initialize *Mathematica* especially for your project, picking up all the different types of resources such as code, Java classes, notebook stylesheets, and palettes that you are developing. If you have divided your work into a set of related projects, the launch can also be made aware of these. There is also a very convenient way to switch between different versions of *Mathematica.*

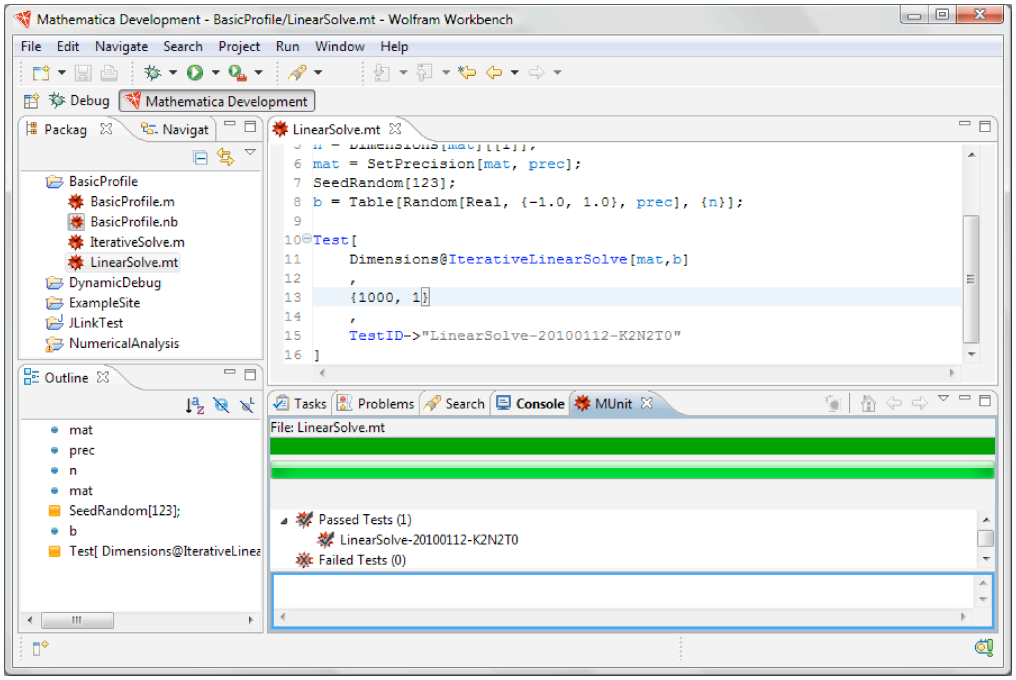
Debugger/Profiler

Workbench provides an interface to the *Mathematica* debugger and profiler. You can inspect many types of *Mathematica* programs, including parallel and user interface code. If you have Java source code in your project, you can debug the *Mathematica* and Java code at the same time—quite a unique feature.



Unit Tester

Workbench provides an interface to MUnit, the *Mathematica* unit tester. You can run individual test files, or whole suites of test files.



Refactoring

Workbench provides a number of refactoring tools for renaming variables and other tasks. It also has specialized search/replace tools that use *Mathematica* pattern language to find and modify code. They work on the actual expression tree with patterns, rather than on the text of the source code with string search/replace tools.

Errors/Warnings

Workbench finds and reports many classes of errors and warnings in all the *Mathematica* source files in the project. This applies whether the source file has been opened or not. The ability to get a quick overview of problems in the source is a tremendous boost of productivity, especially if you have many source files.

TODO Code Documentation

Workbench finds and reports TODO comments in all *Mathematica* source files in the project. This applies whether the source file has been opened or not. This is a simple and powerful way to document and track tasks that need to be carried out in the source.

Source Control

Workbench provides a client to CVS; it can be extended to most other forms of source control.

Take Advantage of *Mathematica*

Mathematica provides a number of features that are particularly useful for software development, but which are often overlooked. Nonetheless, they are all powerful advantages and you should use them frequently.

Easy Serialization of Data

The fundamental building block of *Mathematica* is an expression. Everything, including programs and data, is an instance of an expression. Since expressions can easily be written to a file and then loaded back into *Mathematica*, this makes it very easy to save the state of your system and then load it back again.

This gives an easy way to test and develop your system as components. For example, suppose that you need data from a database; you could develop a Database component and then use this to prepare some data files. Then when you develop your Computation component, you would just load the data and run the calculation. Alternatively, you can write unit tests that load these data files. It is important to make sure you only test one component, and this is a great way to do so.

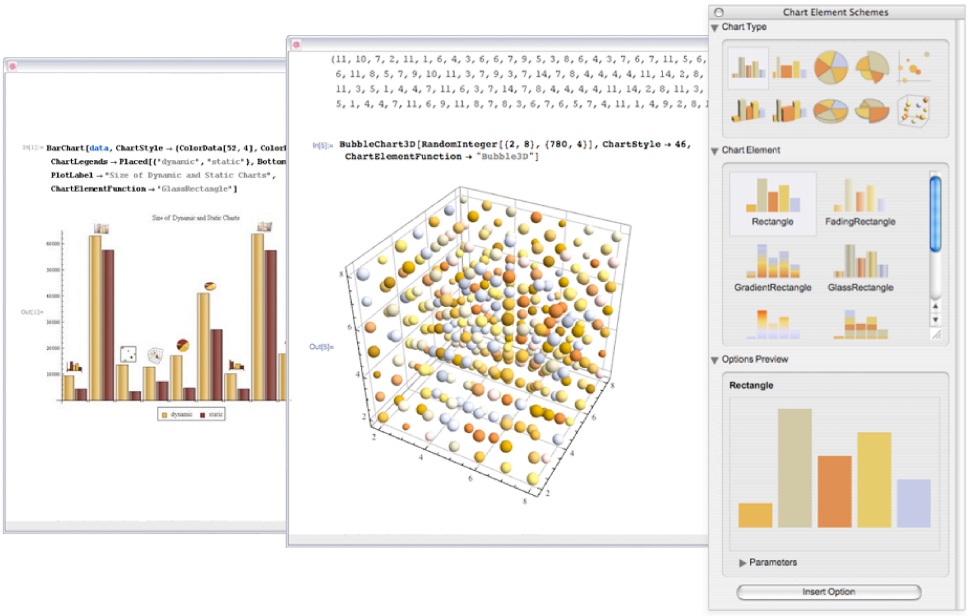
Saving data is a simple process, literally one line such as `Save[file, data]`. If you were working in Java or C++ you would have to write many lines of code, which would have to be rewritten if your data changed. The same applies to loading back into *Mathematica* where a simple `data = Get[file]` restores the data.

Easy Access to Code Points

You can easily call different functions in your packages directly by entering their names into the *Mathematica* notebook front end. This uses *Mathematica*'s interpreted nature and significantly reduces the amount of preparation that is done, for example, before a debugging session as you develop your code.

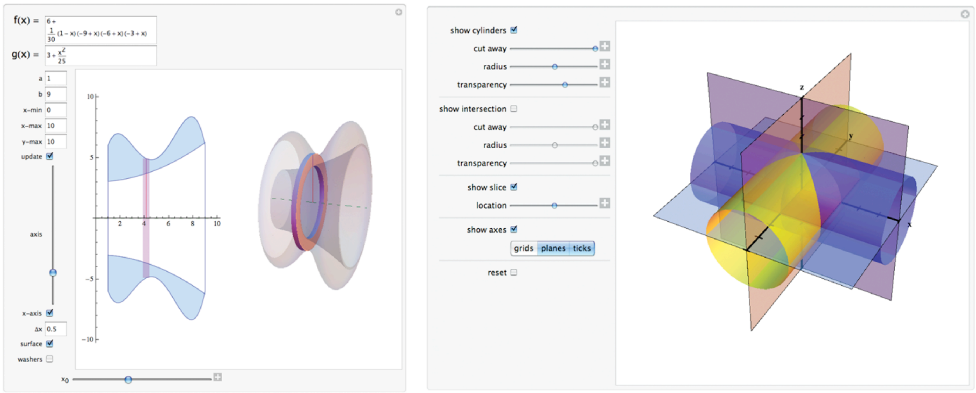
Use Visualization Tools

Mathematica contains many visualization tools for plotting data and surfaces. Some of the tools are quite specialized, such as plotting graphs. All of these can be very useful to give some insight into the running of your application. You can make plots of running time to see if it scales with increasing input size, or you can study the distribution of transformed data looking for unexpected features that might indicate something to investigate.



Explore Interactively

You can use the *Mathematica* interactive tools to explore your functions. For example, in one line you can use the `Manipulate` function to create a tool that will call your function. This might help uncover strange unexpected behavior.



Build Random Models

If your application needs special data to run, you can of course read this from a database. You could use the serialization techniques to store and retrieve the data for developing and testing some of your components. An alternative, especially relevant for structured data, would be to use *Mathematica*'s random number generation techniques to build random data. This can have an additional advantage that the generated data has no real content, so it could be distributed in ways that the real data might be more restricted.

Think of Other Developers

Every developer should always think of other developers. You should follow principles such as documenting code and keeping it simple. This is the case even if you are the only person who ever works on the system. While that might be true at any instant in time, it might not always be the case. Eventually, another developer might be involved. A manager should never let developers treat a part of the code as their own private area where they can suspend the rules of good development practice.

Upgrading Your System

It might be that your project does not follow any of these principles. This can happen if your work grew incrementally from a small start, and you never stopped to rethink how you did development. Perhaps, you would like to change your practices but have not (maybe you are concerned with the cost of a change). Alternatively, you might not see any point in changing, believing that it is all a bit abstract and "does not really apply to me".

Just as hardware and software technologies constantly and rapidly evolve and change, so do software development practices. If you spend any significant time developing software, it is worth spending some of it on a regular basis thinking about how you do this and what new techniques and ideas you can pick up and learn. This document gives some ideas on how you can do this for the case of *Mathematica* development, and it is never too late to improve the way you work and gain a benefit.

If you want to try and apply some of the ideas in this document, you should probably first think about the architecture and component nature of your system. Deciding if your system is already split in components requires a certain amount of judgment. An extreme case would be one large application entirely written in one cell in a *Mathematica* notebook document. In this case your application is probably completely monolithic, with no division into components.

Whatever your actual implementation, you should first review your system and identify its components. Often this will be relatively obvious: for example, your application might have parts that provide a user interface, it might work with a database, and it might carry out some computations. The computations might have some special features: for example, they might need some special way to do interpolation. Each one of these is a potential component.

Once you have identified your components, you can refactor the system to divide it into these components. First, it would be very good to develop a suite of tests so that you can compare your system before and after refactoring. However, the lack of a component nature might be a big impediment. For example, if your application can only be run through the user interface and always has to connect to the database, this will make tests much harder to write and to run. Whatever you do, as you split up your application, you can also make sure that the code uses the appropriate package format if it does not already. You could also locate and use a source code repository.

Moving in this way should get your system into a form where the ideas and practices in this document are useful and relevant, and you can boost your productivity.

Summary

The principles described in this document are really common to all software engineering. But the goal is to show how to do this with *Mathematica*, and so the discussion and the examples show the *Mathematica* side of things. This helps to reinforce the case that *Mathematica* certainly fits into robust software engineering extremely well with up-to-date tools, such as Wolfram *Workbench*, and practices, such as unit testing. Going beyond this, certain features of *Mathematica*, such as expression serialization and graph visualization, lead to some innovative and unique ways to carry out software development. We believe that *Mathematica* is a great system for large software development; following rules and practices and using appropriate tools makes it even better.

Explore Wolfram Online Resources

Let our website help you get started with Wolfram *Mathematica* and *Workbench*.

LEARN MORE ABOUT *MATHEMATICA*

Explore the features, functions, and applications of *Mathematica*.

wolfram.com/mathematica

GET MORE *WORKBENCH* RESOURCES

See videos, download example projects, and search the Workbench documentation.

wolfram.com/workbench

WATCH A VIDEO SCREENCAST

Brief screencasts show you how to incorporate *Mathematica* into your everyday tasks immediately.

wolfram.com/screencasts

FIND A *MATHEMATICA*-RELATED BOOK

The latest *Mathematica*-related books, covering topics as diverse as programming, art, engineering, finance, computer science, and much more.

wolfram.com/books

FIND INSTRUCTIONS IN OUR "HOW TOS"

"How tos" give simple step-by-step instructions to solve specific problems in *Mathematica*.

reference.wolfram.com/howtos

ATTEND A FREE SEMINAR

Free online seminars led by senior Wolfram Research technical staff provide live answers to your questions.

wolfram.com/seminars

READ ONE OF OUR TUTORIALS

Tutorials provide in-depth instruction on using *Mathematica* and how it pertains to your work.

wolfram.com/tutorialcollection

- Package Development
- *Mathematica* Application Development Guide
- The Software Engineering of *Mathematica*
- The Structure of *Mathematica*
- *Mathematica* File Organization

OTHER WEB RESOURCES

- Access full *Mathematica* documentation at reference.wolfram.com
- Explore the web's most popular and extensive mathematics resource at mathworld.wolfram.com
- Post questions, announce sites, and share solutions with fellow users at forums.wolfram.com

WOLFRAMRESEARCH