Introducing your team to an IDE

S. Dawson
Chandler, AZ
stevensveditor@gmail.com

M. Ballance - Mentor, a Siemens Business
8005 SW Boeckman Rd
Wilsonville, OR, 97070
matt_ballance@mentor.com

I. Introduction

Text editors like Vi and EMACS have been replaced by Integrated Development Environments (IDEs) for developing applications in software languages such as C, C++, and Java. IDEs boost code development productivity by putting information from across large codebases at the user's fingertips, prompting users with context-sensitive information, providing navigation based on the semantics of the language, and in general, simplifying the process of working with large and complex software projects.

This paper assumes that the reader understands the productivity gains an IDE brings to an organization and is looking to introduce an IDE to their organization or team. The paper focuses on some of the challenges that need to be addressed to ensure a successful adoption of an IDE into a new organization after a brief overview of some of the benefits an IDE provides.

II. Why do I need an IDE?

Prior to the introduction of System Verilog in 2005, tests written for hardware verification were simple enough that one could argue that the powerful features of an IDE were, at best, not required, and at worst, were overkill for a verification engineer. Today, the complexity of an object-oriented UVM verification environment rivals that of a small-to-medium-sized software application. When reuse of external and in-house verification IP is considered, the complexity of a high-end UVM verification environment may be on-par with a large software application.

As a result of increasing chip sizes, the introduction of class infrastructure to System Verilog and the adoption of verification methodologies such as UVM, IDE adoption by the design and verification community now provides a significant productivity gain. A typical chip or module development environment today is comprised of the following elements:

- *Hundreds to thousands of files*
- *Tens to hundreds of macros*
- *Tens to hundreds of macro instantiations*
- *A deeply nested class hierarchy used by the verification environment*
- *Long compile and run times*
- *Bug tracking and revision control systems*

The features within an integrated development environment go far beyond simple text editing and greatly simplify the control and management of all this data.

III. IDE Benefits

An IDE goes well beyond simple colorization provided by text editors commonly used by HDL design and verification engineers. IDE features make engineers lives easier and improve productivity by integrating several common development tasks inside a single front-end. Being aware of the language syntax allows the IDE to improve productivity by providing contextual support and code completion as the user is writing or browsing through code. An IDE:

- *Provides the ability to do everything that can be done in a text editor today*
- *Supports the same key bindings used by traditional editors*
- *Provides real language syntax awareness, notifying the user of common syntax errors, and reducing errors typically only found during compile and run phases*
- *Provides a method to navigate through the file system efficiently, in most cases significantly reducing the need to understand the entire file system to be efficient*

- *Provides efficient code navigation, allowing the user to jump to module or class source code from within the editor*
- *Provides code completion suggestions including code formatting, module or class instantiation*
- *Takes advantage of documentation standards like Natural Docs to show context-sensitive help when instantiating or mousing over code*
- *Provides browsers to explore the design and class hierarchy*
- *Native support for revision control, allowing file comparisons, checking in and checking out of files*
- *Do common tasks from within the editor including: diffing files, running "Find & Grep", task (TODO) support, link to bug tracking software, compiling and running code*

In addition to the items listed above, an IDE brings features that aren't possible in traditional editors leading to additional productivity improvements including:

- *Using documentation standards (Natural Docs) to make code more readable and maintainable and generating code documentation out of the comments in the code base*
- *Integrated bug and time tracking within the editor allows developers to remain in a common environment for a larger number of tasks.*
- *Taking full advantage of the power of advanced verification constructs like the register model*

The following table compares an IDE (Eclipse) to Vi and Emacs, which are arguably two of the most popular editors used by design and verification engineers today.

TABLE I
EDITOR FEATURE COMPARISON

Description	Vi	Emacs	Eclipse
Code Colorizer	Y	Y	Y
Regular Expression Support	Y	Y	Y
Tag Function	Y[a]	Y[a]	Y
Full Design Awareness	N	N	Y
Design Search (& Replace)	N	N	Y
Code Formatting	N	Y[b]	Y
True Language Syntax Awareness	N	N	Y
Code Error Checking as you Type	N	N	Y
Class Awareness	N	N	Y
Code Completion	N	Y[b]	Y
Design & Class Browser	N	N	Y
Natural Docs Support	N	N	Y
Revision Control Support	N	N	Y
Bug Tracking Integration	N	N	Y
Light Weight	Y	Y	N

[a] Tag support is available but requires significant effort to build and maintain. Not often used.
[b] Limited support via a plug-in

The only IDE with Verilog support that was available when we started exploring an IDE was Eclipse with the SVEditor plug-in in 2008. AMIQ's DVT Eclipse IDE was made commercially available around the same time. Eclipse is an open source, extensible platform that has been expanded to support multiple languages. Today there are several plug-ins available for design and verification engineers supporting System Verilog. These include (but not limited to) SVEditor [1], Sigasi [2], DVT Eclipse IDE [5]. The authors' experience is with the SVEditor plug-in, but the concepts apply to Sigasi, DVT Eclipse IDE and other Eclipse plug-ins. Many of the concepts will apply beyond IDEs not built on the Eclipse framework, though their implementation details may differ.

Over the past 5 years many more options have become available and should be considered when selecting an advanced editor or an IDE. Most commercial editors support System Verilog code colorizers. Advanced editors include VSCode, Sublime and Atom. Additional non-Eclipse based IDEs include SlickEdit and IDEA (JetBrains).

Some items to consider when selecting an IDE:

- *Does the IDE support all the languages I use day to day, not just Verilog?*
- *Does it support my favorite key bindings / colors?*
- *Does the IDE support my revision control system?*
- *What operating systems does it run on?*
- *Are the IDE and plug-ins being actively developed?*
- *What are the hardware (CPU and Memory) requirements of the IDE?*

IV. IDE ADOPTION CHALLENGES

The biggest challenge to switching to an IDE is that "it's different". There will always be a short period when the engineer will be forced to "do it differently". During this time the engineer will feel less productive and as a result will feel some frustration and be tempted to switch back to their previous editor. For most engineers this phase will last about 2-4 weeks.

Engineers are being given the option to switch from something that they have been very productive in, to something that is different. Some of these differences in look and feel, including color schemes and key bindings are easy to overcome or at least mitigate using a set of preference files.

One of the more difficult challenges to overcome is that the engineer will have a fundamental change in the way source code is edited and browsed. Before switching to an IDE, the command line is the central point of focus for the engineer. The command line is used for navigation to browse and select files to edit, used for revision control, to diff files, grep for strings and almost every part of design management.

With an IDE, the IDE takes over many of these tasks, with the IDE being the focal point for the engineer. Files are no longer edited as single, point objects with the engineer navigating to the specific file being edited on the command line. When using an IDE there will only be a single instance of the IDE running. The IDE takes over file management, with the user quickly getting to the point where it is common to have 20 to 100 files open in the IDE at any given time. Navigating through source code, searching for strings, compiling code, navigating to compiled errors all takes place within the IDE.

Many of the advanced code navigation and editing features that bring enormous value and productivity gains also require configuration in the IDE. Traditional editors, of course, require no configuration to the access the more-modest features that they support. To the average user, this setup effort can seem very daunting.

It is often difficult to get engineers to "stick with it" long enough to have them start to appreciate the significant productivity gains that come with using an IDE as many of these engineers are not even aware of some of the advanced features available in their current editor.

And finally, the perception that editing with a simple text editor is 'faster' will need to be overcome.

Your goal is to make the transition to an IDE as seamless as possible, so new users do not give up before getting to understand the power and productivity gains that come from the IDE.

V. INTRODUCING AN IDE TO YOUR ORGANIZATION

This section will focus on Eclipse as an IDE as this has the largest number of plug-ins supporting design and verification engineers. Many of the concepts apply directly to any IDE you may choose to adopt.

A. Getting to know Eclipse

The first thing you will need to do is to get to know how to install and use Eclipse and your preferred development plug-ins yourself. The simplest way to get started is to download and install an IDE distribution that comes pre-configured with a set of plug-ins. DVKit [3] and Sigasi Studio [4], DVT Eclipse IDE [5] are Eclipse distributions that come pre-configured with a plug-in for Verilog and System Verilog. DVKit also includes as a series of plug-ins that support files and activities typically required by design and verification engineers including Vi key bindings, shell scripting, command line interface, Ruby, Python, Perl and TCL, and it's free.

There are several online tutorials that will guide you through some Eclipse basics. The SVEditor website contains a tutorial which walks a user through many of the Eclipse features and is specific to a design verification engineer environment [9].

B. Understand your Environment

Your next step is to provide the IDE with a list of source files for a project that you are working on. The IDE will compile your code, in much the same way your simulator or synthesis engine compiles your source code. This step is required for the IDE to build a picture of the entire environment, and be able to provide context sensitive code completion, class browsing and other advanced features.

To create an argument file, you are going to have to understand your project's source files, and the way they are structured. Try to align the way that source files are specified for the IDE with the way source files are specified for the normal compilation process. This ensures that the IDE doesn't become a burden. Ideally, if the same core file lists can be used to drive design and test bench compilation and drive IDE source indexing, the effort to maintain the core file lists will benefit IDE users.

C. Get Comfortable with the IDE

Use the IDE and identify any short-comings or improvements that are required before rolling it out to the group. Take the time to contact the developer for any issues found during early usage. The developers will typically be very happy to assist you with any question you may have.

Do not roll the tool out to others before you are comfortable working with it yourself. Engineers will be looking for a reason to go back their old editor.

A bad first impression will take years to undo.

D. Eclipse and Plug-in Installer

Now that you are an expert in developing using an IDE, it is time to make adopting an IDE easy for others. While IDE installation seems like an easy first step, it is something that is worth spending time on. Eclipse releases major builds the third Wednesday in June every year plus maintenance releases every 13 weeks. There is also a bewildering array of plug-ins available, often for the same language. Plug-ins are upgradeable and will be upgraded frequently making a read-only installation less than ideal. Different engineers will have different language requirements, some just requiring System Verilog support, others requiring C, Perl, TCL support or some combination of the above.

After some trial and error, we found installing a vanilla Eclipse version and allowing users to add their own plug-ins worked best. We developed a simple GUI that allows the user to select between different language and feature plug-ins. Once the engineer selected plug-ins, the installer installed these via the Eclipse command line. This simple installer greatly streamlined installation for a new user, preempting 99% of common installation issues faced when initially introducing engineers to Eclipse. Provide installation support for both the languages commonly used at your workplace as well as utility plug-ins such as Vi/EMACs key bindings, the revision control system you use and your bug-tracking system. The Eclipse plug-in OOMPH can be used to assist with plug-in and preference management [12] as an alternative to a home-grown installer.

TABLE II
LIST OF USEFUL PLUG-INS

Language / Feature	Plug-in Name
VI Key bindings	VWrapper
EMACS Key bindings	Emacs+
Perl	EPIC Perl
TCL	Eclipse Dynamic Languages Toolkit (DLTK)
Python	PyDev
Bourne and C-Shell Scripts	ShellEd
Command Line Console	TM Terminal
GIT	Installed as a part of Eclipse

E. Develop a Preference File Set

Take the time to create a group of preference files that contain some basic settings. Colorization in Eclipse is extremely flexible, to the point that every language has an independent colorizer. You'll need to spend some time creating a preference file set that has consistent colors regardless of the programming language. Color schemes are highly personalized and are very important to users. Different groups within your organization probably all have the "official" Vi color scheme… and they differ from one other. Create a preference file for each of these. At a minimum have a light color scheme (Emacs) and a dark color scheme (Vi). The Eclipse "Color Theme" plug-in can be used to streamline this process [11].

The default preference file set can be used to include common settings such as your company's decision on Tabs vs. Spaces, specific key bindings that you may need to override due to Eclipse key binding conflicts with your operating system.

F. Automate Project Creation

Eclipse can be used in "single file" mode. In this mode you will not be getting the full benefit of the IDE. Advanced features such as code completion and some other features will either be broken or partially working. To harness the full power of Eclipse it needs to have a full picture of the design you are working on. Project creation can be complex and will need to be automated. Expecting users to go through the project creation process when switching to or checking out a new workspace will stifle adoption. When this step is working correctly the user will simply have to import a project into his workspace to get editing. Project creation covers 3 main topics.

- *Argument files*
- *Resource filtering*
- *Variable definitions*

Eclipse will need an argument file(s) so that it is able to index your code and provide advanced editing functions like code browsing and code completion. In SVEditor argument files are specified in a file ".svproject". This is a simple XML file which is trivial to generate through a script. Sigasi also uses a standard Eclipse project definition file. DVT Eclipse IDE uses a configuration file which is very similar to simulator argument files / command lines.

Resource filtering is useful to limit the scope of files (or resources in Eclipse terminology) that Eclipse will show when opening files. Resource filters are stored in a ".project" file, which like ".svproject", is an XML file which is easy to create. Files hidden by the resource filtering option will be unavailable within Eclipse. To prevent confusion only machine generated files that you wouldn't open in an editor should be filtered initially. Examples include simulator compiled output and waveform files. Make a note of the resource filtering in your training so users aren't caught by surprise.

An Eclipse workspace will often contain multiple projects, where the difference between projects is only in some environment variables. Eclipse has a mechanism to store these as a part of the ".project" file. The SVEditor website has a series of example scripts available for download to give you a leg up on the process of automating project creation [10].

There are plug-in specific project customizations which are also worth investigating including:

- *Company specific naming convention and lint checks*
- *Run configurations*
- *Company specific templates (Copyright headers, directory layout etc.)*

G. Develop Training Material

An IDE is a highly capable program. It has all the features your current editor has, and that's just the beginning! The training material should cover basic editing and should also highlight advanced features that are available out of the box in an IDE. I found that having a hands-on workshop where the engineers are "doing it" as you go through the class is far more effective than a lecture by itself.

The flow for the training developed covered the following items in order:

- *The history of an IDE, and more specifically about Eclipse*
- *Eclipse Terminology (workspace, projects, plug-ins etc.)*
- *Work through launching Eclipse and installing plug-ins via the installer*
- *Import a preference file and how to toggle the Vi / Emacs key bindings on and off. This will* ***make Eclipse feel more familiar early in the class***
- *Work through the steps required to create Eclipse project files (typically "make eclipse", or possibly these files are checked out as a part of the source code)*
- *Import the project created*
- *Basic editing follows*
- *Now it is time to wow the students. At this point you start introducing features that aren't available to design and verification engineers in traditional editors*
- *File and buffer management follow, highlighting how easily Eclipse can be used to navigate through the file system*
- *Advanced features follow this including Class & Hierarchy browsing, Code Navigation, Code Completion, Error checking as you type, Natural Docs etc.*

H. Check on Logistics

You will need to understand and work with your CAD department as you scale IDE usage up through the organization. For example, Java, and by extension Eclipse takes more system resources than a traditional editor. Keep an eye on the memory and resource consumption of the IDE. This is especially true if your engineers are working on shared servers. CAD should be made aware of a possible increase in memory usage for IDE instances. An IDE usage model is to launch and keep it running indefinitely.

Many of the IDEs available have a cost associated with them which will need to be budgeted for.

I. Get Managerial Support

One of the most common reasons for giving up on Eclipse in the first month is that "I have too much work". If you are presenting the class to a specific group, it is often best to start by convincing the group manager that the productivity improvements from an IDE are worth it. If you can get the managers buy in, having the manager publicly allow (insist) that employees use Eclipse for at least a month full time can make a significant difference to the overall adoption rate. At the end of that month, the employees will be making an informed decision on which editor to use going forward.

J. *The Roll Out – Presenting the Workshop*

Your next step is to identify a group of motivated users who are most likely to change to an IDE. I have found that these include engineers who have used an IDE previously (new college grads), users taking over a legacy block (that need to learn it from scratch) and engineers new to the company or group. I have found that engineers new to the group or company have very high Eclipse adoption rates. This group will work with you as you go through the inevitable stumbling blocks that occur as the new tool is introduced to a wider audience.

Present the workshop you prepared to the group. Acknowledge the fact that this is change, and that change is hard. Highlight the fact there are "super user" features that will be available out of the box which will make their lives easier.

As you present the class, don't overlook the power of "simple" text-oriented IDE features such as project wide search or search-and-replace. While less powerful than semantic refactoring features, it is easier to use than UNIX's find and sed.

Don't hide differences between Vi/Emacs and the IDE. It is better to highlight some of the common differences, and to discuss them as differences, neither better nor worse than to let users stumble across them and conclude different is bad.

Throughout the class emphasize that it will take 2 weeks to a month feel as productive as they are today. Highlight that after this period they will naturally start taking advantage of the advanced features which will **make their lives easier** than they are today.

At the end of the workshop be prepared with print-outs of commonly used key bindings [8]. Have a print out that shows the steps to create and import a project as well as preference files.

VI. Optimizing Eclipse

On smaller, macro level designs no work will need to be done, Eclipse will work right out of the box without doing anything special. As the design gets larger, Eclipse may become less responsive. The following section describes some common items that will make Eclipse run more smoothly.

K. *Memory Envelope*

If Eclipse starts to become less responsive, the first thing to do is to ensure that the Java virtual machine has enough memory to work in. The default setting in the eclipse.ini file is 1Mbyte which on larger projects is insufficient. This can easily be set to a larger value by either updating the eclipse.ini file in your installation or adding the -Xmx switch on the Eclipse command line when launching. You can monitor the Eclipse heap size via Window>Preferences>General>Show Heap Status

L. *Workspace Location*

These second item to look at is any caching done by language-editing plug-ins. Plug-ins typically store the compiled image (index) of the code inside the user's workspace. Eclipse will interrogate and update the index of compiled information as code is edited. If Eclipse is running on a stand-alone laptop, the workspace is local to the machine and no optimization needs to be done. If running Eclipse on a server, it is possible that the workspace will be stored on the local network. At this point Eclipse will be accessing the index via the local network when updating or querying the index. Keeping the workspace on a disk that is physically on the same server as Eclipse is running on makes a significant difference on larger projects.

M. *Automatic Indexing*

Evaluate whether turning off automatic indexing updates make a difference. Eclipse will automatically index files as you type by default. This can be turned off, preventing the momentary pauses that can occur as the index is queried. If this is turned off, the user will have to manually re-compile from periodically.

Disable Eclipse scanning the drive for updates by disabling Preferences>General>Workspace>"Refresh using native hooks or polling"

N. *File / Module Exclusions*

Consider excluding files or sections of files that do not add value to a user editing experience from the project file. For example, a verification engineer may not care about thousands of RTL files, which can be excluded and will be treated as a black box by Eclipse. Standard cell and pad libraries are other good candidates for exclusion.

VII. SUMMARY

An IDE brings a modern programming experience to Design & Verification engineers. Introducing your organization to an IDE will provide some short-term challenges and requires up front planning. IDE adoption rates and retention rates will be significantly improved by developing a hands-on workshop, creating configuration files and automating project creation. Once your team starts using an IDE, they should be as productive as they were in under a month. After the initial adjustment period, productivity gains will increase as the team adopts more of the advanced features that the IDE brings.

VIII. REFERENCES

[1] SVEditor Website, https://sveditor.org
[2] Sigasi Website, http://insights.sigasi.com
[3] DVKit Installation Site, http://dvkit.org/
[4] Sigasi Studio Installation Site, https://www.sigasi.com/download-b/
[5] AMIQ DVT Eclipse IDE Website, https://www.DVT Eclipse IDE.com
[6] Atom Editor, https://atom.io/
[7] Sublime Editor, https://www.sublimetext.com/
[8] Key bindings, https://sites.google.com/site/svedvkit/user-guide-documentation/key board-shortcut-summary
[9] Eclipse and SVEditor Tutorial, https://sites.google.com/site/svedvkit/getting-started/tutorial
[10] Project Automation, https://sites.google.com/site/svedvkit/user-guide-documentation/sveditor-user-guide/automating-project-setup, https://github.com/sigasi/SigasiProjectCreator
[11] Eclipse Color Theme, http://www.eclipsecolorthemes.org/
[12] Eclipse OOMPH, https://projects.eclipse.org/projects/tools.oomph