



---

# IceCube Software

---

---

## *An Introduction to the IceCube Software Development Environment*

---

**Simon Patton L.B.N.L.**

This document provides an introduction to the concepts and tools that make up the IceCube Software Development Environment. While concentrating on the “Personal” aspect of this environment it also gives an overview of how this system works in conjunction with the “Collaborative” environment.

## 1.0 Introduction

---

### 1.1 Purpose

The purpose of this document is to introduce the concepts and tools that make up the IceCube Software Development Environment. However, it is not intended to be a authoritative reference on any of these tools. That information can be obtained from their individual documentation. (See 1.3 “References”)

The intended audience for this document is anyone who will be using the IceCube Software Development Environment. This includes, but is not restricted to, developers and users of “production” IceCube software. While it is not required that developers of non-production IceCube software use this environment, every effort has been made to make the environment easy to use and such people are encouraged to try it out.

### 1.2 Scope

The scope of this document is to provide enough information about the concepts and tools that make up the IceCube Software Development Environment so that someone who is new to this environment can get up and running and make productive contributions to the “production” IceCube software. To this end this document will also discuss how the tools introduced here help with the implementation of the software process that should be followed for all “production” software.

This document concentrates on the “personal” software development environment, as that contains those components of the IceCube Software Development Environment that an individual developer will most commonly use. However this document will also outline the “collaborative” software development environment so that an individual developer can see how the two different environments interact and to highlight those parts of the collaborative environment that provide feedback to a developer on the state of their code in the context of the complete “production” IceCube Software effort.

As noted earlier this document is not intended to be a authoritative reference on any of these tools.

### 1.3 References

The following references are used throughout this document.

[MMM] The Mythical Man Month. Fred Brooks

[IceSoft-xxx] IceCube Code Guidelines.

[IceSoft-yyy] BFD User’s Guide.

[IceSoft-zzz] FAYE User’s Guide.

### 1.4 Definitions and Acronyms

*baseline*: An agreed set of code and resources that function together.

*head*: The version of a project which contains the “highest” production tag, plus any modification which have been made on top of that tag.

*package*: A Java concept which defines a naming and accessibility scope.

*project*: A collection of Java class files and resources that are all packaged into a single jar file.

*release*: A “baseline” which has been given to the “clients”.

*workspace*: A directory in which, or below which, the process of developing a feature or fixing a bug is executed.

### 1.5 Overview of this Document

The next section gives an outline of the benefits of a common development environment to a Collaboration. It also outlines how the components that can make up such an environment can be assigned to one or both of two distinct categories, namely the personal and collaborative environments.

Following that section the rest of the document is divided into two Parts. The first Part covers the components of the personal environment. It gives the details needed to use these components, both in a standalone mode and in a baseline mode. The baseline mode is one used to develop “production” IceCube software.

The second Part covers those components of the collaborative environment which do not appear as part of the personal environment. This Part expands how these interact with the personal components and what a developer should keep an eye upon to make sure that their software is behaving correctly in the context of the complete “production” software effort.

---

## 2.0 A Common Software Development Environment

---

As Fred Brooks outlined in his seminal work [MMM], as the number of people on software effort increases linearly, the number of line of communications increases geometrically. This means that, very quickly, it is very hard to keep track of everything needed to successfully contribute to an effort. Therefore one of the ways for a large software effort to succeed is for it to use common mechanisms for tasks such as configuring, building and testing the code and, most importantly, communicating the past, current and future state of the effort. It is these mechanisms which constitute a common software development environment.

This section will discuss the benefit that can be gained, both by a collaboration and by an individual developer, from the use of a common software development environment. It will also outline the responsibilities of such an environment in more detail.

The general benefits that can be gained for such an environment are the following:

- It facilitates efficient communications between developers.

As all developers can effectively talk the same language they do not have to spend time clarifying details that are not directly relevant to the issues at hand, namely design and maintenance of the software product.

- More time can be spent on developing the product, rather than having each developer set up their own environment.

If the environment is already available, then the developers of a product do not have to spend their time creating on, and if one does not exist only a small number of developers need commit time to developing one for everyone else to use.

- It is easier for developers to transition to other parts of the software effort.

When a software effort has a common environment that means that the tools and policies a developer learns when working in one area of the effort and immediately applicable when they have to move to a different area of the effort

The next few sub-sections look at the areas of code configuration, build and testing in more detail.

### 2.1 Configuring Code

When a software effort involves more than one file or more than one developer there is a need to know which files work together and what changes have been made to those files. To this end the software development environment should needs to include some sort of “Software Configuration Management” tool. This will allow an individual developer to use and, if permitted, define sets of files that function together. A set of such files is often referred to as a “baseline”. A developer normally develops new features and corrects bugs against a specific baseline. The idea of a “baseline” set of files is also used to define the files that make up a final software product. In these cases the features and performance of the entire software product are tested before the baseline is finally defined and in this case the baseline is normally called a “release”.

### 2.2 Building Code

Building on the concept of baselines, it would be really useful for developers to be able to use pre-build files, such as libraries, that have been built for the set of files that make up a baseline, rather than each developer having to create and maintain such a set of files. Therefore the common build system can offer such a facility. This means that developers can concentrate on their contribution of the product, rather than spending time worrying about whether the rest of the files they use are correct.

Having a common scheme for building contributions also means that it is easy to automate the building of the complete software product. Automation of this complete build means that it can be easily done at regular intervals, for example every night, and thus problems in the current version of the product such as incompatibilities between individual contributions, can be detected at an early stage of development and corrected.

### 2.3 Testing Code

The benefits raised in the preceding section about having a single approach to building code also apply to having a common testing scheme. Moreover, having such a scheme helps with the whole development and maintenance cycle.

On an individual level, testing allows a developer to see whether their contribution is performing as expected. It also lets them check that none of the changes they have introduced during development have broken any of the features that were already working. This helps them avoid submitting code which will need to be revisited at a later date. On a collaborative level, testing is the easiest way to demonstrate that all the individual contributions are functioning together correctly in the completed product and that there are no unexpected side effects from combining individual contributions.

Having a testing scheme also means that when there are problems with a product, it is easy to isolate which area is not behaving as expected and then that problem can be assigned to the correct developer. This helps to stop the other developers from being distracted from their development by hunting for the problem in their contribution. This also allows them to continue to work, by using a baseline which tests indicate does not contain the problem.

### 2.4 Personal and Collaborative Environments

The use of a common software development environment not only means that developers can easily collaborate, but it also means that when all the individual contributions are brought together into a completed software product, a single system can be used to validate it. As has been noted in the preceding sections, the process of validating such a product has different requirements than the process of creating the individual contributions of that product, therefore a common software development environment can be divided into those components necessary for one or both processes. This document will refer to those components which a developer uses to create an individual contribution to be elements of the “personal” software development environment, while those used to validate the entire product will be referred to the “collaborative” environment.

Part I of this document will cover the elements of the “personal” software development environment, while Part II will cover those components that are part of the “collaborative” software development environment, that have not been covered in Part I.

# Part I: The “Personal” Software Development Environment

---

## 3.0 The Aim of a “Personal” Software Development Environment

---

The aim of a “personal” software development environment is to provide an individual developer with all the tools they need to create, test and submit code that is part of a larger collaborative software effort. Of course this does not exclude such an environment from being used to create a non-collaborative software product. In fact one of the subsidiary aims of this “personal” software development environment is to help the developer used “good” development practices and as such even a non-collaborative software product can benefit from using this environment.

As noted in Section 2.0, a software development environment should provide tools to help configure, build and test contributions to a software product. To this end the “personal” software development environment should include tools that allow a developer to do the following:

- define a baseline upon which they can base their work;
- have access to pre-built files, i.e. libraries, in that baseline (to save space and time during development);
- build their contributions against those pre-built files, rebuilding any files which are dependent on their changes if required;
- run their own tests in the same environment as the collaborative tests;
- access to a code repository where they can submit their changes for archiving;
- submit their changes for inclusion in the “next” baseline;

One area that is specifically excluded from these requirements is the prescription of a code editor. The reason for this is simple. Providing that the code produced by the developer conforms to the coding guidelines laid out in IceSoft-xxx and all the other artifacts needed by the code, e.g. `build.xml` files, correctly function in the collaborative environment, then it does not matter how a developer edits their code. Therefore the choice of editor (and possibly IDE) can be left up to the developer.

While this software development environment is designed to be used in a collaborative environment and exploit the idea of baselines, it is easier to walk through a standalone example, i.e. one that does not use baselines, to give developers a feel for the environment and development process before introducing the extra complications of a baseline. The next section is just such a walk through.

## 4.0 A Standalone example of the PSDE

---

All work in a development environment is done in or below a directory that is referred to as the “workspace”. While it is not a requirement that work on each feature or bug take place in its own workspace, experience shows that this is a wise approach. That way the set of changes associated with each task can easily be separated, which is especially important if one set turns out to be flawed.

### 4.1 Preparing a Workspace

In IceCube the workspace should be created by hand by doing a simple directory creation and the changing into that working directory.

```
[patton@acme]$ mkdir work
[patton@acme]$ cd work
```

Baseline management and access to the source code repository are tightly coupled responsibilities, therefore there is a single tool, the “Baselined File Development” system (`bfd`) which is used to handle these responsibilities. If the IceCube software is already installed on your machine, and you have executed the standard IceCube initialization then you should already have access to this tool. The easiest way to check is the following command:

```
[patton@acme]$ which bfd
```

This will either give you some information about where it is installed or tell you that there is no such command, in which case you should refer to Appendix A.

When running in a standalone mode `bfd` needs access a tools directory which contains the third party software products that IceCube is using. Again this directory should already be installed by the standard IceCube installation, but if is not then Appendix B details how such a directory can be set up by hand. In either case the following command can be used to initialize a standalone workspace, where the final argument of the command is the location of the `tools` directory:

```
[patton@acme]$ bfd init /home/icecube/tools
```

The workspace is now ready for use. However before using the workspace there are some environmental variables needed by the tools. These can all be set by the following command for `sh` users:

```
[patton@acme]$ . setup.sh
```

or the following command for `csh` users:

```
[patton@wallaby]% source .setup.csh
```

### 4.2 Using Existing Projects

A “project” is a collection of source and resource files that are all packaged into a single library. In the case of Java this will be a jar file. This is the basic unit of development and each project is the responsibility of a single developer or, in some of the large cases a small team of developers. A project should not be confused with the Java idea of a “package” which defines naming and access scopes.

In order to access an existing project either because that is the project you need to work on, or the project you are working on depends on it and you are not using the baseline mode of `bfd`, then you need to make sure that your `CVSROOT` environmental variable is set to the relevant repository and then issue the `checkout` option of the `bfd` system. This option copies all the source and resource files needed to build a project into the current workspace. In that case of the `preston` project which we will create in, it will eventually depend upon the `icecube`, `faye` and `icecube-faye` projects. To gain access to these projects, when using the standalone mode of `bfd`, you will need to use the following commands.

```
[patton@acme]$ bfd checkout icebucket
[patton@acme]$ bfd checkout faye
[patton@acme]$ bfd checkout icecube-faye
```

After that you will need to “build” these projects. That is the topic of the next section.

### 4.3 Building a Project

“Building” a project means creating some or all of the files necessary to use and test a project. A typical Java based project will generate two files, the `<project>.jar` and `<project>-test.jar` files. The first of these jar files contains all the executable code and resources needed to use the project, while the second jar file contains all the code and resources needed to test that the project is functioning correctly. To create these two jar files you simply need to execute the `lib` target of the project’s `build.xml`. You can execute any `ant` target for a particular project in a one of the following ways.

- You change into the project’s main directory and run `ant` there; or
- run `ant` in the workspace directory specifying the project to use; or
- set the default project and then run `ant` in the workspace directory.

The following examples, all of which begin with the workspace as the current working directory, demonstrate these options. The first example demonstrates the first option, along with showing which jar files appear in the `lib` directory of the workspace:

```
[patton@acme]$ cd icebucket
[patton@acme]$ ant lib
[patton@acme]$ cd ..
[patton@acme]$ ls -l lib
```

The next example demonstrates the second option, again including the display of the generated files:

```
[patton@acme]$ ant -DPROJECT=icebucket lib
[patton@acme]$ ls -l lib
```

The third option is demonstrated in the following third and final example:

```
[patton@acme]$ ant -DPROJECT=icebucket defaultProject
[patton@acme]$ ant lib
[patton@acme]$ ls -l lib
```

Once all three projects have been built, we can look to creating the `preston` project.

#### 4.4 Creating a New Project

In the “production” environment the usual way of creating a new project is to submit a request to the IceSoft librarian and, if approved, they will let you know when the new project’s area is ready for use. When not working in the production environment you may or may not wish to archive your code. If you do not want to archive you code then you can simply create a new directory for the project in your workspace and proceed. If you do want to archive you code outside the production environment it is assumed that you will be responsible for you own archive and so you will be able to create a new directory for a project within that archive. (Appendix C has details on how you can do this using CVS.)

To create the standard contents of a new Java project using bfd, you simply need to execute the following commands, where preston is the name of the newly created project:

```
[patton@acme]$ bfd checkout preston
[patton@acme]$ ant -DPROJECT=preston defaultProject
[patton@acme]$ ant -DPACKAGE=icecube.preston.wool \
-Dpackage_dir=icecube/preston/wool createProject
```

This demonstrates the normal use of the `defaultProject` target, namely to set up the project one which you are working as the default one, and thus keep most ant commands simple. The `PACKAGE` parameter sets the name of the default Java package into which any new class or interface will be placed unless explicitly stated otherwise. The `package_dir` parameter is a kludge which will be obsolete once we integrate an ant task to translate a package name into its directory path. Until then it must be specified by hand and the value of `package_dir` should be the same as `PACKAGE` but with the “.” replaced by “/”.

Figure 1 shows the contents of the preston project directory all of which are created by the `createProject` target. The `build.xml` file is used by ant to build the output files of the project (both the `<project>.jar` and `<project>-test.jar` files and well as the Javadocs documentation). The properties file is used to set project specific values that are used in any ant build. It is unlikely that you’ll need to edit the `build.xml` file as that is standard, but you may need to edit the properties file to declare dependencies of the project (See 4.5.2 “Declaring Dependencies”).

---

**FIGURE 1:** The contents of the preston project after it has been created.

---

```
preston
+---+ build.xml
+ project.properties
+ src
|   +---- icecube
|   |       +---- preston
|   |       |       +---- wool
|   |       |       |       +---+ package.html
|   |       |       |       + test
|   |       |       +---- package.html
+ resources
   +---- test
```

---

The `src` directory is where source files are kept. These can be both code and documentation source files. Files in this directory will not normally be included in the output jar files of a project. Conversely, the `resources` directory hold those files which should be included as part of the output jar files. Files in the `resources/test` directory are automatically included in the `<project>-test.jar` file, while all others are included in the `<project>.jar` file.

Within the `src` tree you can see that two `package.html` files have been created in the `wool` and `wool/test` directories. These are skeletons for documenting how these two packages should be used. It is recommended to keep these files as up to date as possible, i.e. include changed as soon as they are made.

## 4.5 Creating a New Class

To begin the development of the preston project we are going to create a “Plug-in” for FAYE which will simply count the number of Event records processed when the program is run. To do this we need to create a new class in the default package of the preston project. This is done using the `createClass` ant target, as demonstrated by the following command.

```
[patton@acme]$ ant -DCLASS=Counter createClass
```

This creates the following two files:

```
preston/src/icecube/preston/wool/test/CounterTest.java
preston/src/icecube/preston/wool/Counter.java
```

The first file contains a skeleton for the unit tests for the new class, while the second file contains a skeleton for the class itself.

### 4.5.1 Defining the Tests for a Class

While this is not the place for a detailed discussion on software process or unit testing, the recommended approach for code development is to write the unit test first. Yes, the tests should be written first. The detailed benefits of this approach are described elsewhere but in essence writing the tests first documents the expectations for a class and also it minimizes the chances of feature creep. show the modifications to the skeleton `CounterTest.java` file that need to be made so that it can correctly test the `Counter` class when it is written.

---

**FIGURE 2:** The Import statements to be added to `CounterTest.java`

---

```
import faye.Stop;
import faye.test.MockSyncValue;
import icecube.faye.frame.IceCubeStream;
import icecube.faye.test.IceCubePlugInAdapterTest;
```

---

**FIGURE 3:** The new inheritance declaration for the `CounterTest` class

---

```
extends IceCubePlugInAdapterTest
```

---

---

**FIGURE 4:** The private static data to be added to the `CounterTest` class

---

```
/** An event stop to cause the event() method to be called */
private static final Stop[] EVENT_STOPS =
    new Stop[] {new Stop(IceCubeStream.EVENT,
        new MockSyncValue(3)),
        new Stop(IceCubeStream.EVENT,
            new MockSyncValue(4))
    };
```

---

---

**FIGURE 5:** The definition of the `setUp` method for the `CounterTest` class

---

```
protected void setUp()
{
    testObject = new Counter();
    setAdapter(testObject);
    super.setUp();
}
```

---

---

**FIGURE 6:** The declaration and definition of the `testEventCount` method for the `CounterTest` class, which replaces the `testSomething` method

---

```
/**
 * Tests that the Counter PlugIn increments its count correctly.
 */
public void testEventCount()
{
    testConfigureFromReady();

    prepareFrameSupplied();

    Assert.assertEquals(testObject.getCount(),
        0);
    testObject.frameSupplied(getFrameToken(),
        EVENT_STOPS[0]);
    prepareFrameSupplied();

    Assert.assertEquals(testObject.getCount(),
        1);
    testObject.frameSupplied(getFrameToken(),
        EVENT_STOPS[1]);

    Assert.assertEquals(testObject.getCount(),
        2);
}
```

---

As `Counter` is going to be a plug-in for the FAYE it is different for a simple Java class in that its test code, rather than inheriting from the `TestCase` class, inherits from the `IceCubePlugInAdapterTest` class. For more details on creating and testing FAYE plug-ins see `IceSoft-zzz`. The purpose of the rest of the code modifications should be reasonably clear.

The new `CounterTest.java` file is now ready for compilation, however this will not work for the following reasons:

- The compilation dependencies for the file have not been added into the system.
- The `Counter` class has not been declared yet so can not be resolved.

### 4.5.2 Declaring Dependencies

Dealing with the dependency issue first, `ant` and the JVM need to know upon which jar files `CounterTest.java`, or more correctly the `preston-test.jar` file, is dependent. This is done by editing the `project.properties` file in the main directory of a project. From the import statements in Figure 2 we can see that the test jar file has dependencies on the following files:

- `faye.jar`
- `faye-test.jar`
- `icecube-faye.jar`
- `icecube-faye-test.jar`

Therefore these file names need to be added to the `all.dependencies` and `test.dependencies` entries in the properties file. Note that these entries have different separators, the “all” list uses a comma, while “test” list uses a space. As `preston` does not depend on the logging system, this is also a good juncture to remove the `log4j` declarations in this file. Figure 7 shows how the modified lines should appear in the new version of the properties file.

---

**FIGURE 7:** The modified dependency lines in the `project.properties` file.

---

```
all.dependencies = faye.jar,faye-test.jar,icecube-faye.jar,icecube-faye-test.jar
,junit.jar
project.dependencies =
test.dependencies = faye.jar icecube-faye.jar faye-test.jar icecube-faye-test.jar
../tools/lib/junit.jar
```

---

### 4.5.3 Defining the new class

Now that the test for the `Counter` class has been created we can address creating the class itself. Looking at the requirements `Counter` must satisfy if it is going to execute the tests successfully we can see that it needs the following features.

- A public default constructor.
- An implementation of the `event` method of the `IceCubePlugIn` interface.
- A `getCount` method which returns the current number of times the event method has been called.

Figure 8 to Figure 11 show the modifications to the `Counter.java` skeleton that are needed for this class to conform to the interface requirements outlined above, while

---

## A Standalone example of the PSDE

---

Figure 12 shows the necessary changes to the `project.properties` file to account for the import statements in Figure 8.

---

**FIGURE 8:** The import statements to be added to `Counter.java`.

---

```
import faye.Stop;
import icecube.faye.IceCubePlugInAdapter;
import icecube.faye.frame.IceCubeFrame;
```

---

---

**FIGURE 9:** The inheritance declaration for the `Counter` class.

---

```
extends IceCubePlugInAdapter
```

---

---

**FIGURE 10:** The public default constructor for the `Counter` class, which replaces the original private one.

---

```
/**
 * Create an instance of this class.
 */
public Counter()
{
}
```

---

---

**FIGURE 11:** The declaration of the `event` and `getCount` methods for the `CounterTest` class.

---

```
/**
 * Increments by one the number of EVENT stops processed by this PlugIn
 */
public void event(final IceCubeFrame frame,
                 final Stop stop)
{
}

/**
 * @return the number of EVENT stops that have been processed by this
 * PlugIn
 */
public int getCount()
{
    return 0;
}
```

---

---

**FIGURE 12:** The modified dependency line in the `project.properties` file.

---

```
project.dependencies = faye.jar icecube-faye.jar
```

---

At this point in time it should now be possible to at least compile the new class and its tests using the following command (executed in the main directory of the workspace).

```
[patton@acme]$ ant lib
```

---

## A Standalone example of the PSDE

---

While this has now built the code into two new jar files, these files are not very useful as they do not contain the functionality needed, and thus the tests will fail. To prove this the tests can be run using the following command.

```
[patton@acme]$ ant test
```

As you can see there is one failure. To be able to get more details on this failure we can create a test report with the following command.

```
[patton@acme]$ ant report
```

As the report target is the default one for ant we can get the same results from the following command.

```
[patton@acme]$ ant
```

The generated report is in the form of HTML files. Therefore to view the results you need to open the following file in a Web browser of your choice.

```
docs/junit/index.html
```

Once you've done this and navigated through to the failure you'll discover that it has the following message:

```
expected:<0> but was:<1>
```

This is correct as the `getCount` method currently always returns zero. The "fix" for this is to actually implement the interface which we declared for the `Counter` class. Figure 13 and Figure 14 show the modifications necessary to achieve this.

---

**FIGURE 13:** The private data to be added to the `Counter` class.

---

```
/** The number of EVENT stops that have been processed by this PlugIn */  
private int count = 0;
```

---

---

**FIGURE 14:** The definition of the `event` and `getCount` methods for the `CounterTest` class, (Javadocs comments have been omitted for brevity.)

---

```
public void event(final IceCubeFrame frame,  
                 final Stop stop)  
{  
    count++;  
}  
  
public int getCount()  
{  
    return count;  
}
```

---

Now running the following command

```
[patton@acme]$ ant
```

should produce success and a clean bill of health in the final report.

## 4.6 Archiving Development

(This section is only applicable if you are working on production code, or you are archiving your own code development. If you are not archiving your own code development then you can skip this section.)

There are stages during the development of a project where it is useful to archive the progress made so far even though the development may not have been completed. This can be done using the `archive` option of the `bfd` system. This will commit the current version of the project into the code repository. However before we can do that we have to declare to the `bfd` system those which files should be archived. In the case of the `preston` project we need to add all the files we have created so far.

To add a file to the `bfd` system for archiving you simply need to use the `add` option. This can be done with the following list of commands.

```
[patton@acme]$ bfd add preston/build.xml
[patton@acme]$ bfd add preston/project.properties
[patton@acme]$ bfd add preston/resources/test
[patton@acme]$ bfd add preston/src/icecube/preston/wool/Counter.Java
[patton@acme]$ bfd add preston/src/icecube/preston/wool/package.html
[patton@acme]$ bfd add bfd add \
preston/src/icecube/preston/wool/test/CounterTest.Java
[patton@acme]$ bfd add bfd add \
preston/src/icecube/preston/wool/test/package.html
```

For clarity's sake each file has been added individually in the above example, however it is also possible to specify all the files to be added in a single `add` line.

Now that these files have been added we can archive them with the following command.

```
[patton@acme]$ bfd archive -m "New project files" preston
```

The `-m` option provides a short message about the files being archived. If this is omitted then an editor will be displayed so that a longer message can be composed.

It should be noted at this point that, in the `bfd` system, archiving files does not mean that they immediately become part of the continuous build system, which is covered in more detail in Part II. To make that happen the project needs to be "delivered". The next section deals with this operation.

## 4.7 Delivering a Project

Once the development of a feature within a project has been completed all the necessary files need to be flagged and delivered to the next stage of development which could either be sub-system integration testing or entry into the continuous build system. Any delivered project is expected to pass all of its own tests and not cause any of the larger-scale tests to fail. If a delivery does not satisfy these requirements then it must be backed-out.

Delivery of a project to the next stage of development is done by the `deliver` option of the `bfd` system. When delivering a change it is necessary to either specify a tag that will uniquely identify the change or specify the type of change in which case a new "production" tag will be generated automatically ("production" tags take the form `VXX-YY-ZZ`). This means that one of the following options must be used to deliver a package.

- `-t <tag>`: This option is used to provide a private `tag` or a production tag that can not be generated by one of the following options.
- `-b`: This option is for the delivery of a “bug” modification. Delivery of a “bug” modification normally increments the `ZZ` portion of a production tag<sup>1</sup> and implies there have been no changes to the public API of the project.
- `-n`: This option is for the delivery of a “minor” modification. Delivery of a “minor” modification increments the `YY` portion of a production tag and implies that there have been some changes to the public API of the project, but the main feature set remains unchanged.
- `-j`: This option is for the delivery of a “major” modification. Delivery of a “major” modification increments the `XX` portion of a production tag and implies that there have been significant changes to the public API of the project, and that the main feature set has been changed.

In the case of `preston`, the new public class means that there has been a change in the public interface. However, as the development of this version of `preston` is not complete, for instance it would be useful to add code that allows the counter to be reset, this initial version should be delivered as a minor modification using the following command.

```
[patton@acme]$ bfd deliver -n preston
```

Which gives `preston` the `V00-01-00` production tag. It should be noted that when a project is delivered, its local copy is set to match the new tag. This has a number of consequences, the most obvious is that it is not possible to announce new files in a project. New files can only be added to the “head” of a project, where the “head” is defined as the “highest” production tag plus any modification which have been made on top of that tag.

To be able to add files to a project it is necessary to use the “head” version of the project. This can be done using the following command.

```
[patton@acme]$ bfd head preston
```

### 4.8 Finishing with a Workspace

As discussed at the top of this section, it is recommended that work on each feature or bug take place in its own workspace. This means that once the work has been completed the workspace should be disposed of to allow it to be used for other work. The following command<sup>2</sup> checks that it is safe to dispose of a workspace and then removed all the files, so that the directory can be recycled as another workspace if it is so desired.

```
[patton@acme]$ bfd dispose
```

- 
1. The other possibilities will not be discussed here but details can be found in `IceSoft-yyy`.
  2. This feature is not yet implemented so the workspace currently needs to be “disposed” of by hand. A `rm -fr *` command will do this, but make sure you are in the workspace *before* you contemplate such an option.

### 4.9 Summary

The `bfd` (baselined file development) system is the tool used to by the personal software development environment handle file management. This section has been a simple example of using the personal software development environment, while using the `bfd` system in standalone mode. The following `bfd` option have been introduced:

*init*: initializes a workspace for standalone mode.

*checkout*: copies the contents of a project into the workspace.

*add*: declares that a file should be added to a project.

*archive*: copies the working version of all project files into a code archive.

*deliver*: signals that the current version of the project is ready for the next integration step.

*head*: update a project to the “most recent” version.

*dispose*: clean up a workspace.

This section also introduced some of the `ant` targets that are in the standard workspace `build.xml` file:

*lib*: creates the library files for a project from that project’s source files.

*defaultproject*: sets the project that is used in a workspace if no other is explicitly specified.

*createProject*: creates the skeleton files needed to create a new project.

*createClass*: creates the skeleton file needed to create a new class,

*test*: run the all the tests for a project.

*report*: create an HTML report on the results of a project’s tests.

Also discussed in this section is how dependencies are declared in the `project.properties` file.

The next section discussed how this simple example changes when the `bfd` system is used in baseline mode.

### 5.0 A Baseline example of the PSDE

---

...To be written...

---

---

# Part II: The “Collaborative” Software Development Environment

---

6.0

---

---

## Appendix A: Installing the bfd System by Hand

---

The `bfd` system should be installed when the IceCube software distribution is installed on a machine, along with the necessary environment settings. However if this is not available then this appendix outlines how this system can be installed by hand.

To install a local copy of `bfd`, start by moving to the directory in which you want it placed. The following is an example of where you may want to place the files:

```
[patton@acme]$ cd $HOME/bin
```

Next you need to make sure that the right environmental variables are set up to access the IceCube repository. For `sh` users the following commands will do this

```
[patton@acme]$ export CVSROOT=:ext:acme.lbl.gov:/home/icecube/cvsroot
[patton@acme]$ export CVS_RSH=ssh
```

For `csh` users the following are necessary:

```
[patton@wallaby] setenv CVSROOT :ext:acme.lbl.gov:/home/icecube/cvsroot
[patton@wallaby] setenv CVS_RSH ssh
```

After these are set you need to “export” the `bfd` project from the repository into the current directory with the following command:

```
[patton@acme]$ cvs export -D tomorrow bfd-tools
```

(The `tomorrow` option is used to guarantee that you get the latest version of the code.)

If you have an `ssh` key set up on `glacier.lbl.gov` then you should not need to do anything extra to get all the `bfd` files. However if you do not have such a key installed you will be prompted for your password, which you should provide.

Now you need to set up the necessary links and variables to run the `bfd` program. This requires setting up a soft-link, placing this link into your `PATH` variable and specifying a “root” directory. For `sh` users these tasks can be done at the prompt using the following:

```
[patton@acme]$ ln -s bfd-tools/bfd.py bfd
[patton@acme]$ export PATH=${PATH}:${HOME}/bin
[patton@acme]$ export BFDROOT=/usr/icecube/bfdroot
```

Meanwhile `csh` users need to execute the following:

```
[patton@wallaby] ln -s bfd-tools/bfd.py bfd
[patton@wallaby] setenv PATH ${PATH}:${HOME}/bin
[patton@wallaby] setenv BFDROOT /usr/icecube/bfdroot
```

(Note that if `~/bin` is already in your `PATH` variable you do not need to execute the line which adds it to this variable). These environmental settings will be copied into the appropriate setup files that are created by the `bfd` system when it initializes workspace.

The details of the contents of the `BFDROOT` are covered in `IceSoft-yyy` and are only relevant if the baseline mode is being used or there are some local custom-tailoring.

The `bfd` system should now be ready for use.

---

---

## Appendix B: Installing the IceCube tools Directory by Hand

---

Access to the third party tools that are used by the IceCube software is done via the `tools` directory of either the workspace or a baseline. The IceCube software distribution contains all of the third party distributions needed by IceCube software. However if this is not installed then this appendix outlines how the tools can be installed by hand.

Currently, the default `bfd` scripts, and thus IceCube's, are set up to handle to following packages:

*Ant*: a dependency and build system.

*JUnit*: a unit testing framework for Java.

*Xalan*: an eXtensible Stylesheet Language Transform processor.

*Log4j*: a complete logging solution for Java.

*Jython*: an interactive scripting tool for Java.

This appendix covers the versions that were in use in September 2002. In most cases a simple change in the version numbers where they appear is sufficient to make these instructions work for the current versions. The following file, specified with respect to the directory in which the `bfd-tools` directory was created (See Appendix A: "Installing the `bfd` System by Hand"), contains the list of the current versions:

```
bfd-tools/std/scripts/tools_vers.sh
```

Before downloading the tools used by IceCube a suitable directory needs to be created, something like `/home/icecube/tools` is recommended. By convention a subdirectory named `packaged` should also be created, which will be the destination directory of the packaged tools files when they are downloaded. The tools should now be downloaded into the packaged directory. Table 1 to Table 5 give the download information each of the tools.

---

**TABLE 1:** Download information for ant 1.5

---

Web Site	<a href="http://jakarta.apache.org/ant/">http://jakarta.apache.org/ant/</a>
Download URL	<a href="http://jakarta.apache.org/builds/jakarta-ant/release/v1.5/bin/jakarta-ant-1.5-bin.tar.gz">http://jakarta.apache.org/builds/jakarta-ant/release/v1.5/bin/jakarta-ant-1.5-bin.tar.gz</a>

---

**TABLE 2:** Download information for JUnit 3.7

---

Web Site	<a href="http://www.junit.org/">http://www.junit.org/</a>
Download URL	<a href="http://download.sourceforge.net/junit/junit3.7.zip">http://download.sourceforge.net/junit/junit3.7.zip</a>

---

---

---

**TABLE 3:** Download information for Xalan 2.4.0

---

Web Site	<a href="http://xml.apache.org/xalan-j/">http://xml.apache.org/xalan-j/</a>
Download URL	<a href="http://xml.apache.org/dist/xalan-j/xalan-j_2_4_0-bin.tar.gz">http://xml.apache.org/dist/xalan-j/xalan-j_2_4_0-bin.tar.gz</a>

---

**TABLE 4:** Download information for Log4j 1.2.3

---

Web Site	<a href="http://jakarta.apache.org/log4j/">http://jakarta.apache.org/log4j/</a>
Download URL	<a href="http://jakarta.apache.org/log4j/jakarta-log4j-1.2.3.tar.gz">http://jakarta.apache.org/log4j/jakarta-log4j-1.2.3.tar.gz</a>

---

**TABLE 5:** Download information for Jython 2.1

---

Web Site	<a href="http://www.jython.org/">http://www.jython.org/</a>
Download URL	<a href="http://prdownloads.sourceforge.net/jython/jython-21.class?use_mirror=unc">http://prdownloads.sourceforge.net/jython/jython-21.class?use_mirror=unc</a>

---

Once downloaded each tool should be installed. For those files with the `.tar.gz` suffix the following command, when executed in the `packaged` sub-directory, will unpack and install them:

```
[patton@acme]$ gunzip -cf <file> | tar -C .. -x
```

Files with the `.zip` suffix can be handled with the following command:

```
[patton@acme]$ unzip -d .. <file>
```

While to install Jython you need to execute the class file, which can be done with the following command:

```
[patton@acme]$ java -cp . jython-21 -o Jython-2.1 demo lib source
```

(Note that in this case the `.class` suffix should *not* be specified.)

Now all the necessary files should be in the chosen tools directory and so you can proceed with the use of the `bfd` system.

---

---

## Appendix C: Creating a New Project using CVS

---

When working on non-production code it is advisable to maintain a personal code archive. At present CVS is the choice of archive for IceCube and so this appendix gives details on how you can create you own archive and create a new project with that archive. (It should be noted that you need version 1.11 of CVS or later to be able to use multiple archives transparently!)

Creating a CVS archive is very easy. All you need to do is make the root directory for the archive and initialize it. The following commands do just that for `sh` users.

```
[patton@acme]$ mkdir ~/cvsroot
[patton@acme]$ export CVSROOT=~/cvsroot
[patton@acme]$ cvs init
```

Users of `csh` can use the same sequence of commands, replacing the `export` line with the following.

```
[patton@wallaby] setenv CVSROOT ~/cvsroot
```

Having created the archive all you need to do to create a new project is to make its directory. For example, for the preston project used throughout this note the following command will be sufficient.

```
[patton@acme]$ mkdir ${CVSROOT}/preston
```

Don't forget to set up the `CVSROOT` environmental variable to point to the correct archive.