Pavlov Code Conventions

1. Pavlov for Developers

As more people start working with the Pavlov code, it's important to recognize and maintain the strengths in the code and to attack its weaknesses. If something in this document doesn't make sense to you, that's OK -- do a quick web search to try to get an explanation. If that doesn't work, feel free to post a message on the Pavlov forums.

2. Tools

The following tools are necessary or useful to have installed on your computer:

- Java Development Kit JDK1.5.0B1 or later (needed to build Pavlov)
- Ant (needed to compile Pavlov)
- A CVS client of your choice (needed to check code in and out)
- JUnit (only needed to write/run tests)
- A reasonable text editor or integrated developement environment

> **Note:**
> If you don't intend to recompile Pavlov (i.e. if you aren't a programmer), you can get JRE1.5.0B1 or later -- the Java Runtime Environment. This is much smaller than JDK1.5.0B1.

3. Readable Code

Let's start with the following code conventions:

1. Avoid special characters (smart-quotes, copyright signs, etc) at all costs
2. Indent your code so that it's readable
3. Avoid going over 80 characters per line
4. Label problem areas in code with `"//FIXME: Description of problem"`
5. Name temporary variables better than TJ does

The following depend on your IDE. When Ant's checkstyle task can handle tiger (jdk1.5) code, it will become much easier to enforce.

- Avoid uploading source with TAB characters
- Avoid uploading source with MS-DOS linebreaks ("\r")

4. Good Code

Try to adhere to these rules as much as possible. If you see an instance where a rule is violated, at least add a `FIXME` or else fix the problem. Many of these come from Joshua Bloch's book "Effective Java Programming Language Guide."

1. [Bloch01 #47] Don't ignore exceptions
2. [Bloch01 #12] Minimize the accessibility of classes and members
3. [Bloch01 #23] Check parameters for validity.
4. [Bloch01 #38] Adhere to generally accepted naming conventions
5. [Bloch01 #29] Minimize scope of local variables
6. [Bloch01 #28] Write doc comments for all exposed API elements
7. [Bloch01 #34] Refer to objects by their interfaces
8. [Bloch01 #33] Beware the performance of string catenation
9. [Bloch01 #15] Design and document for inheritence or else prohibit it
10. Use final variables and parameters where possible.
11. Use tiger-style looping as possible
12. Use tiger-style typechecking as possible
13. Strive to decrease `javac -Xlint` warnings
14. Write unit tests as possible
15. Use assertions to check preconditions and postconditions
16. Use log4j instead of `System.out.println`
17. If a method doesn't depend on the object's state, make it static

5. Good Documentation

Even with a great deal of recent effort, Pavlov's documentation is not in great shape. Check out How To Write Doc Comments for Javadoc to learn how to do it right. Apart from JavaDocs, it's good to have good, normal comments in your code, like:

```
070  /**
071   * Returns the next question answered fewest times or null if problem.
072   *
073   * @return a <code>QuestionData</code> value
074   */
075  public QuestionData getQuestion(Vector<QuestionData> exclusion) {
076
077    QuestionData q = null;
078
079    int lowest = 10000000; // FIXME: should be Integer.MAX_VALUE
080
081    for(QuestionData x: questions) {
082        if(exclusion!=null)
083            if(exclusion.contains( x)) // if this question is in the exclusion vecto
084                continue; // go to next question
```

```
085         if (x.getTotal() < lowest) {  // this one's been asked fewest times
086             lowest = x.getTotal();
087             if(lowest==0) return x;
088             q = x;
089         }
090     }
091     return q; // question with fewest answers or null
092   }
```

6. Hey! Let's look at that example!

Since it's sitting there, let's make some comments on it.

- Line 71, "if problem" should be explained better
- Line 73, not bad, has return value documented
- Line 75, uses tiger style type checking on the vector (rule 12)
- Line 75, exclusion isn't changed by this method, should be a final parameter (rule 10)
- Line 79, the FIXME reminds us to come back to that later (a randomly selected big number is not as good as the biggest number possible)
- Line 81, tiger style looping (rule 11), good
- Lines 82-83 should be combined into one if statement
- Lines 82-84, if blocks should have braces (C programmers do this all the time)
- Lines 85-86, should calculate `x.getTotal()` once and store in (final) temp var (rule 10)
- Line 75, could substitute `AbstractList` for `Vector` ((tricky) rule 7)

This is what "code cleaning" is about, and while it's not as glamourous as some other aspects of development, it cuts down on bugs, makes the code faster, and shows everybody that you're smarter than the original developer. :)

7. Design

Object Oriented Design (OOD) isn't a newbie topic, but an understanding of some design patterns will make parts of the Pavlov code easier to understand. You can read about these on the Web, in the famous "Gang Of Four" book "Design Patterns", or [Stelting02]. The following patterns are often used.

- Strategy (aka Method-Object). Example: user.BasicStrategy.
- Template Method. Used all over the place.
- Observer (aka Listener). Example: event.AnswerListener.
- Model-view-controller. Almost the entire user interface.
- Abstract Factory. main.AbstractTemplateKit is one example.
- (Static) Factory Method. Used often.
- Builder. main.standalone.SwingUIBuilder aspires to be a real Builder.
- Iterator. Many, many classes use iterators.

- Callback. Example: LoginController calls AbstractPavlovApplication back.
- Composite. Used often.
- Plugin (aka Create-A-Bunch-Of-Strategy-Classes-From-Files-In-A-Directory-At-Runtime-Pattern). Probably a dozen cases of this throughout the code, from skins, to themes, to question selection strategies, to feedback pluglets.

8. Baby Steps

To many programmers, this document, not to mention the size of the Pavlov code base may seem overwhelming. When you make a task for yourself, try to make it small enough to handle in one session. For example the task "I'm going to decrease the scope of all the variables in pavlov" is too big. I often pick a subpackage, like pavlov.user or pavlov.library and bang on it, compile, get rid of Xlint warnings, run the program for a while, then check my code in. The longer your code is different than the CVS repository, the more of a chance of a check-in collision (count yourself lucky if you don't know what that means) or breaking the build. These problems are bound to happen, but we should try to minimize them as much as possible.

9. Keep it Friendly

I started working on Open Source projects in about 1990. Some projects were great because the developer community was talented and supportive of each other -- everybody learned a lot, had fun, and made cool software. In others, conflicts arose and things got nasty. (I remember one morning when a developer at Lawrence Livermore National Laboratory got a 80-gigabyte email for breaking a build.) As this community grows, I require that it remain courteous. On the other hand, sometimes people won't be able to help in what you feel is a timely manner, and you may feel offended. One of the things that project work like this teaches aspiring programmers is to be thick skinned and self reliant.

10. Recommended Environment

If you're a seasoned developer, you probably have strong views about what development environment you use. I don't have any interest in getting into an argument like "xjpicovim rules and winusurper sucks!" But, if you are starting out from scratch, I can make a couple of suggestions.

Install linux! (I'm almost kidding.)

Linux is a great environment, it's free-no-cost and free-liberty. Blah, blah, blah. It takes some work and some disk space to get it installed and it takes a while to get used to it. But in the long term, its worth it.

But, you want to get started hacking out code quick. Cygwin-X provides a set of tools that make your computer "feel like" Linux with XWindows and it's pretty painless to install. You can get all the tools you need to work with sourceforge (ftp, ssh, scp, cvs) and great time-tested editors like XEmacs and vim. Getting to know these tools, especially if you're a college student (in math, physics, computer science, engineering), is really, really useful.

There are some open-source Java IDE's that are probably worth looking at, but I don't have an informed opinion on them. At the least, they should provide text highlighting, auto indenting, and CVS support.

Note:

Since I wrote this, I've downloaded and installed jEdit. When its plugins begin to support JDK1.5 better, I'll be able to wholeheartedly recommend it. It rocks for JDK1.4 code. It's also very good for XML code (such as editing Pavlov books.)

11. Bibliography

These are books I've drawn (heavily) on in preparing this document.

[Bloch01] Bloch, Joshua. *Effective Java Programming Guide.* Addison-Wesley, Boston, MA, 2001.

[Stelting02] Stelting, Stephen and Maassen, Olav. *Applied Java Patterns.* Sun Microsystems Press, Palo Alto, CA, 2002.

[Fowler04] Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D. *Refactoring: Improving the Design of Existing Code*. O'Reilly and Associates. 2004.

[Simmons04] Simmons, Robert. *Hard Core Java*. O'Reilly and Associates. 2004.