Design and Implementation Conventions for Java Programs

1. Define each public class in one .java file. The only exception to the one-class-per file rule is for homogeneous collection or container classes. For such classes, a single .java file may (but is not required to) contain the definitions for both the container class as well as the elements that it contains. For example, a collection class and the protected type of its elements may be defined in a single .java file. If the element type of the container has data members that are defined as other classes, all of those other class must be defined in separate .java files.

2. Format and document .java files *precisely* as follows, with strict adherence to the format of the comments, including the number and placement of *'s and the amount of indentation.

```
import ... ;

/****
 *
 * The comment at the top of a .java file is a high-level description of the
 * class defined in the file.  Start the description with the words "Class X"
 * and then describe the purpose of the class and the major methods it
 * provides.  The method descriptions in this header comment are generally
 * brief, e.g., "Class X provides methods to add, delete, change, and find
 * its elements."  Do not list all of the method details in the header
 * comment, since full comments for each method appear below in the body of
 * the class, at the cite of each method declaration.  The header comment can
 * describe the data representation used in the class in high-level terms if
 * it's germane to explaining what the class is for.  The header comment does
 * not describe low-level details of the data representation or any details of
 * method implementation.
 *
 *
 * @author Name and current email address of file's author; name is at
 *         least first and last, with middle name or initial if necessary or
 *         commonly used by author; email address appears in parentheses
 *         following full name; email address may be abbreviated to a local
 *         address if the full address can be expected to be known to.  E.g.,
 *         Gene Fisher (gfisher@thyme); John H. Smith (john_smith).
 *
 */

[public] class X extends Y implements Z {

    /**
     * Prose description of the method, describing what the method does,
     * not how it is implemented.  Describe the use of each parameter by name,
     * refer to the instance object as "this", and use the word "return" to
     * describe the return value if there is one.  Also describe each data
     * field used as an input and each data field modified as an output.
     * Modification to a parameter or data field includes indirect
     * modification to the value of a reference parameter or data field.
     * Stylistically, use complete sentences, avoid passive voice.
     *
     * pre: formal precondition
     *
     * post: formal postcondition
     *
     */
    public T1 methodName(T2 t2, ..., Tn tn) [throws ...] {
        /*
         * Include comments for each local variable and comments above each
         * line or group of lines that describe how the method works.  All
         * but the "totally obvious" lines of code should be commented.
         * Comments for variables should be descriptive noun phrases.  Comments
         * for code lines should be in complete sentences.
         *
         * All code comments should be formatted exactly as as this one is: (1)
         * start with "/*" on a separate line, indented to the current level of
         * code indentation; (2) start each comment line with "*", indented to
         * current indentation + 1; (3) end with "*/" on a separate line,
         * indented to current indentation + 1.
         *
         * See below under "indentation and spacing conventions" for further
         * discussion of the format of code within a method body.
         */

    }

     ... other public methods ...

    Note no public or private data fields


    Protected methods in same format as public methods.

    /** Comment describing data field */
    protected T1 varname;

     ... other protected data fields ...
};
```

3. Indentation and spacing conventions.
   a. Except for the specific indentation shown within a class definition above, indentation is every 4 spaces.
   b. Wherever possible in a program editor, use tab characters for indentation, and set the tab width to 4 characters. If 4-character tab stops are not supported in a given editor, set whatever indentation parameters are available to be as close as possible to 4-character tab spacing.
   c. The following is a template for indentation and spacing of Java method bodies; blank lines are significant.

```
T1 methodName(T2 t2, ..., Tn tn) {

    /* Comment ... */
    Tv1 tv1;
    ...

    /** Comment ... */
    Tvm tvm;

    /*
     * Comment ... .
     */
    for (...) {

        /*
         * Comment ... .
         */
        if (...) {
            ...
        }
        /*
         * Comment, if necessary.
         */
        else if {
            ...
        }
        ...
        /*
         * Comment, if necessary.
         */
        else {
            ...
        }
    }

    /*
     * Comment ... .
     */
    while (...) {
        ...
    }

    /*
     * Comment ... .
     */
    for (start-expression; while-expression; end-expression) {
        ...
    }

    /*
     * Comment ... .
     */
    for (long-start-expression ... ;
            long-while-expression ... ;
                long-end-expression) {
        ...
    }


    /*
     * Comment ... .
     */
    switch (...) {
        /*
         * Comment, if necessary.
         */
        case c1:
            ...
            break;
        ...
        /*
         * Comment, if necessary.
         */
        case ck:
            ...
            break;
    }
}
```

In the preceding commenting convention, the '//' style of comments can be used instead of the '/* ... */', but the two styles cannot be mixed. For multi-line comments with '//', format as follows:

```
//
// Comment ...
//
```

4. Class and method naming conventions.
   a. The names of MVP model types (i.e., class names) should be identical to the names of corresponding RSL objects. If the RSL object name does not obey the capitalization conventions for type names given below, then the RSL name should be changed accordingly.
   b. The names of model methods should be identical to the names of corresponding RSL operations, except the first letter of all method names should be lower case. If the RSL operation name does not obey the capitalization rules for method names given below (except for first letter lower case), then the RSL name should be changed accordingly.
   c. The names of view types (i.e., class names) should be the same as corresponding model names, with the suffix "UI" added. If a single design defines two or more alternative views, the view class names should be disambiguated with the type of UI for each. E.g., for model class PersonDatabase, view classes PersonDatabaseButtonUI and PersonDatabaseMenuUI are button-style and menu-style UI's, respectively.
   d. The names of view methods should be full-word mnemonic, including multi-word where appropriate, with the same capitalization conventions as for model type and method names. See "capitalization conventions" below for further details.
   e. The names of process types and methods should be full-word mnemonic, including multi-word where appropriate, with the same capitalization conventions as for model type and method names. See "capitalization conventions" below for further details.
   f. The names of constants should be mnemonic, typically one word or two words, all uppercase, with multiple words separated by underscores. E.g., LENGTH, MAX_SIZE.
   g. The names of data fields access methods should start with "get".
   h. The names of data field setting methods should start with "set".
   i. The names of boolean-valued query methods should start with "is".
   j. The names of searching methods should start with "find".

5. Capitalization conventions.
   a. Type names (i.e., class names) should begin with a capital letter, and each distinct word in the name should be capitalized, e.g., `PersonRecord`. If an abbreviation is used in the name, all letters in the abbreviation should be capital, e.g. `PersonDB`.
   b. Method names should follow the same conventions as type names.
   c. .java root filenames must be the same as the class name, including capitalization.

6. Variable naming conventions (including method parameters and class data fields).
   a. For variable names of user-defined types:
      i. Variable names (including parameter and data field names) start with a lower case letter and otherwise follow the conventions for class names.
      ii. A variable name may be a "proper" abbreviation of the type name, e.g., PersonRecord personRec, PersonDB personDB.
      iii. A "proper" abbreviation is an abbreviation the characters of which are a proper subset of the characters in the words of the variable's type. E.g., PersonDB personDataBase is not a proper abbreviation.
      iv. Short variable names are acceptable as long they are sufficiently mnemonically understandable in their context of use, e.g., PersonRecord pr, PersonDB pdb. Given the "proper abbreviation" rule, a variable

name may be no shorter than the total number of capital letters in its type name.

v. When a variable name abbreviates two or more words, the second words and beyond are capitalized.

vi. These rules are one of the few places in these standards open to subjective interpretation. E.g., in a context where few other variables start with the letter "p", the name "pRec" could be sufficiently mnemonically significant to be used instead of the longer "personRecord".

vii. In all cases, variable naming should be consistent throughout a project.

viii. In the case where two or more variables of the same type are declared in a given scope, the names should be suffixed with unique integers starting with 1, e.g., `PersonRecord personRec1, personRec2` or prefixed or suffixed with short mnemonic identifiers, e.g., JTextField nameTextField or JTextField nameJtf.

b. For variable names of atomic types or external library types that follow some other type naming conventions:

i. Variable names (including parameter and data field names) should be lower case, short, and mnemonic, e.g.,

```
boolean status
int counter
```

ii. Where very short names are sufficiently mnemonic, such as with loop counter variables, single-character variables names are allowed, e.g., int i; for (i=1, ...) ... ;

iii. The same numeric suffix and disambiguation rules above should be used.

7. Size limits.

a. No method may be longer than 50 lines, except as noted below. The 50-line length does not include comments and blank lines. This rule may not be achieved by condensing lines in such a way as to violate any of the preceding formatting conventions. E.g., the following is legal

```
for (i=1; i<=n; i++) {
    if (i % 2) {
        x.Foo1(i);
        x.Foo2(i);
    }
}
```

the following is NOT legal

```
for (i=1; i<=n; i++) { if (i % 2) { x.Foo1(i); x.Foo2(i); } }
```

nor is even the following

```
for (i=1; i<=n; i++) {
    if (i % 2) {
        x.Foo1(i);
        x.Foo2(i);}
}
```

The following are exceptions to the 50-line rule:

i. View `compose` methods whose bodies are generated by a GUI builder.

ii. Other segments of code for which the author can rigorously justify the reason to exceed the 50-line rule, where "rigorously justify" does not mean "I'm too lazy to break it up into method calls".

b. No class may have more than 25 public methods plus 25 protected methods, i.e., no more than 50 methods total. If a class has fewer that 25 public methods, it may have up to 50 methods total, however it may never have more than 25 public methods. Typically classes should have far fewer than 50 methods total.

c. No class may have more than 50 protected data fields.

d. Note that combining the 50-line rule with the 50-method rule means that no .java file can be longer the 2500 lines of code, excluding comments. Typically, .java files should have far fewer than 2500 lines.

8. On the use of interface builders for View classes.

a. The overall design of view classes must following the view design conventions discussed in 206 lecture notes 4 through 6.

b. This means in particular that code generated by GUI builders must be placed in appropriate constructor and compose methods.

c. As noted above, the 50-line rule need not apply to the code generated by a GUI builder, as long as the code is placed in an appropriate constructor or compose method.

d. In general, interface builders do not allow direct editing of generated code, which means they are good for generating the initial version of View-class methods, which is then hand edited outside of the interface builder to produce code that meets the 206 design and implementation conventions.

9. Encapsulating non-conventional external libraries and utilities.

   a. Resources provided by any external Java or C++ library that violates any of the preceding conventions must be encapsulated in *exactly one* class.

   b. That is, a "wrapper" must be placed around all non-conventional outside services.