

# EXPERIMENT IN EVOLUTION COMPLEXITY: INSTRUCTIONS TO SUBJECTS

Technical report CSM-431, ISSN 1744-8050 (June 2005)  
Department of Computer Science, University of Essex

Amnon H. Eden

*Department of Computer Science, University of Essex*  
and  
*Center For Inquiry*

**Abstract.** We describe an experiment whose purpose is to establish our hypothesis regarding *evolution complexity*, namely, the relative “flexibility” of particular design policies towards particular changes. We describe a program with two implementations, and instruct the subjects to carry out changes in each implementation, measuring the time each change required.

**Table 1.** Instructions sheet

- ◆ Please read carefully this *Instructions sheet* from beginning to end before anything else.
- ◆ The purpose of this experiment is to test the hypothesis put forth under the title “evolution complexity”. There is no need for you to understand the theory or read Eden and Mens [05]; in fact, the integrity of the experiment is insured by ignoring the predictions we make.
- ◆ Generally speaking, our hypothesis regards the time that certain tasks will take to complete. To test this hypothesis, we ask you to carry out the tasks described in the Tasks tables (Table 2, Table 6, Table 7) to measure the time it took you to complete each using the Diary (Table 8), and to summarize the results in the Summary sheet (Table 9).
- ◆ **Before** conducting this experiment, make sure you that, along with this document, you are sitting next to a computer running a clear display of the digital clock, and that same computer has a working Java environment including a complete installation of Java, including the Java compiler and run-time environment.
- ◆ **Before** you begin working on a particular task, please start a new entry in your Diary: write the number of task under TASK and the current time under START TIME.
- ◆ **After** completing a session, whether you decide to take a break or have finished a task, enter the current time in your Diary under END TIME. When you resume working, start a new Diary entry.
- ◆ **After** completing all tasks, summarize the time it took you to carry out each task and write the total in the Summary sheet (Table 9).

*Sample Diary*

The following three entries in a Diary tell us you spent two 40-minute sessions on task no. 1 and one hour session on task no. 2:

TASK	START TIME	END TIME	LENGTH
1	1-Jan-05, 9:12	9:52	0:40
1	1-Jan-05, 12:20	13:00	0:40
2	2-Jan-05, 23:11	3-Jan-05, 1:11	1:00

*Comments:*

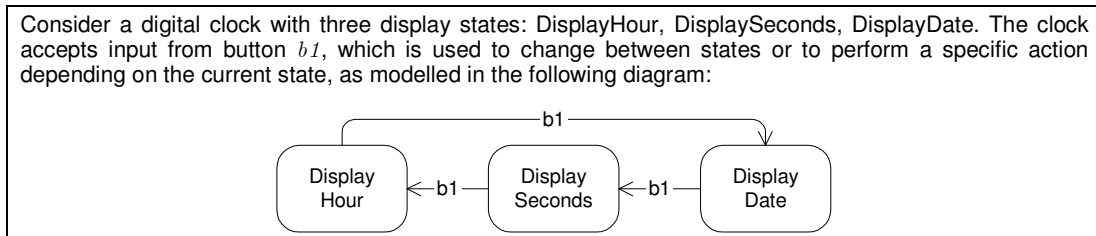
- ◆ Please carry out the tasks in the order specified.
- ◆ Note that every task that involves programming also requires some testing. Note that the testing is part of the task. Therefore, you should include the time it took you to test and correct each change in the respective task.
- ◆ Take as many breaks as necessary during the experiment. It is recommended that you take relaxation rests between tasks: drink tea/coffee or play with your cat.
- ◆ It makes no difference which computer you use or which Java environment you code your program with, as long as the same computer/environment are used for the duration of the experiment.
- ◆ Please do not concern yourself with “scoring well”. This experiment is not meant to test your skills. Take as long as it is necessary to complete each task.
- ◆ If you have any questions at any point, please include the time it takes to correspond or discuss your questions in calculating the time it took to complete the task.

*Thank you for participating in this experiment and good luck!*

**Table 2.** Preliminary Tasks

<b>(Task 1)</b>	<p>Read and understand the <i>State</i> design pattern (Appendix).</p> <p>Please read the entire chapter from beginning to end, including the sample code, before engaging in other tasks.</p>
<b>(Task 2)</b>	<p><u>Implement a Clock using the <i>State</i> pattern</u>: Read the description of the Clock problem (Table 3) and implement a small package conforming to the State pattern as sketched in Table 4.</p> <p>Note that there is no need for you to keep track of the actual time nor to change it. Just implement this solution in the simplest possible way.</p> <p>Upon completion of the programming, write a small test to ensure that each time a “button is pressed” the appropriate action is taken, for example by printing a message on the current time and current state to the standard input.</p>
<b>(Task 3)</b>	<p><u>Implement a Clock in procedural style</u>: Read the description of the Clock problem (Table 3) and implement a small package conforming to the State pattern as sketched in Table 5.</p> <p>Again, there is no need to keep track of the actual time nor to change it. Just implement this solution in the simplest possible way.</p> <p>Upon completion of the programming, write a small test to ensure that each time a “button is pressed” the appropriate action is taken, for example by printing a message on the current time and current state to the standard input.</p>

**Table 3.** The Clock problem



**Table 4.** Object-oriented implementation to the Clock

Use the State pattern to implement the solution to the Clock problem: Define a separate class for each state and use a context object to switch between the states. Specifically, write your program along the lines of the following style (note: “style”, this is not a complete program!):

```
class ClockContext {
    private int hours, minutes, seconds, dayInMonth, month;
    private ClockState currentState;
    public SwitchTo(ClockState nextState) {
        currentState = nextState;
    }
}
interface ClockState {
    void b1(ClockContext context); // button 1 pressed
}
class DisplayHour implements ClockState {
    public void b1(ClockContext context) { /* implement button 1 pressed */ }
}
class DisplaySecond implements ClockState {
    public void b1(ClockContext context) { /* implement button 1 pressed */ }
}
class DisplayDate implements ClockState {
    public void b1(ClockContext context) { /* implement button 1 pressed */ }
}
```

**Table 5.** Procedural implementation to the Clock

Implement a procedural solution to the Clock problem: Define a Java class containing an enumeration of all the possible states, along the lines of the following style (note: this is not a complete program!):

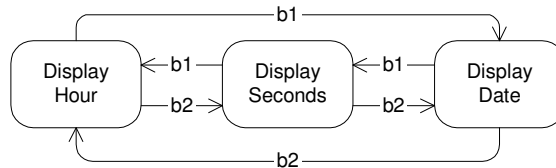
```
class ProcClock {
    enum states = {DisplayHour, DisplaySecond, DisplayDate, SetHour, SetDate};

    private states currentState;

    public void b1() { // button 1 pressed
        switch (currentState) {
            case DisplayHour: /*...*/;
            case DisplaySecond: /*...*/;
            case DisplayDate: /*...*/;
        }
    }
}
```

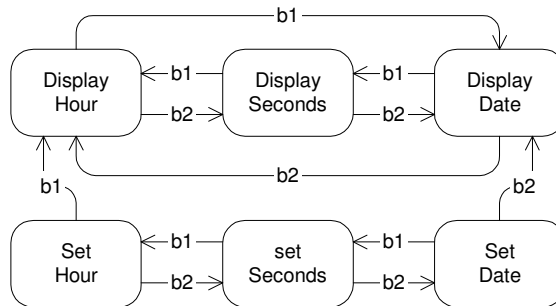
**Table 6.** Tasks for evolving the *State* implementation

**(Task 4)** Add button 2 to the *State* implementation: Add another button to the *State* implementation, *b2*, so as to conform to the following illustration:



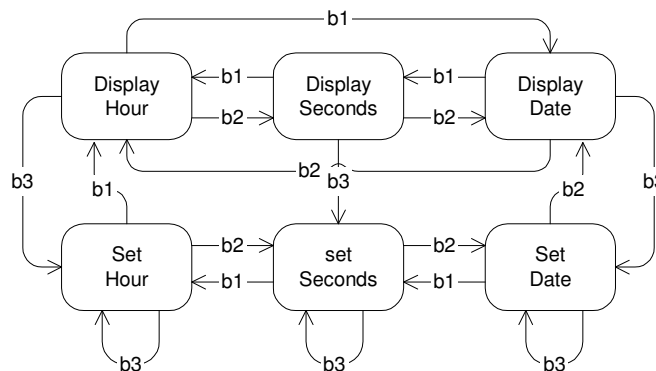
Upon completion of the change, write a small test to ensure that whenever a “button is pressed” the appropriate action is taken, for example by printing a message on the current time and current state to the standard input.

**(Task 5)** Add three states to the *State* implementation: Add the states SetHour, SetSeconds and SetDate to the *State* implementation so as to conform to the following illustration:



Upon completion of the change, write a small test to ensure that whenever a “button is pressed” the appropriate action is taken, for example by printing a message on the current time and current state to the standard input.

**(Task 6)** Add button 3 to the *State* implementation: Add a third button to the *State* implementation, *b3*, so as to conform to the following illustration:

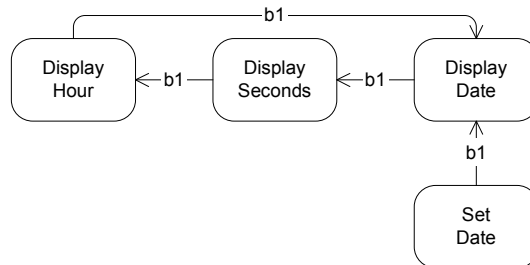


Note that in SetHour (SetSeconds, SetDate) state, pressing *b3* results with increasing the minute (seconds, date) with 1.

Upon completion of the change, write a small test to ensure that whenever a “button is pressed” the appropriate action is taken, for example by printing a message on the current time and current state to the standard input.

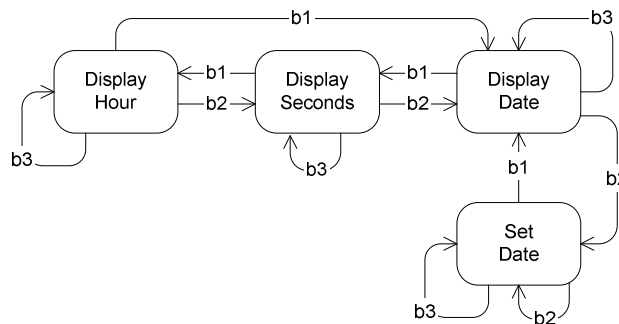
**Table 7.** Tasks for evolving the procedural implementation

**(Task 7)** Add a state to the procedural implementation: Add another state to the procedural implementation so as to conform to the following illustration:



Upon completion of the change, write a small test to ensure that whenever a “button is pressed” the appropriate action is taken, for example by printing a message on the current time and current state to the standard input.

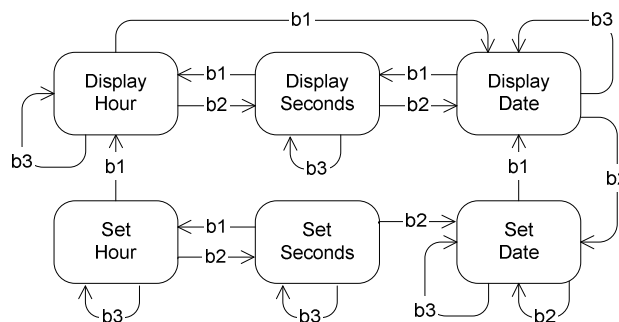
**(Task 8)** Add two buttons to the procedural implementation: Add second and third buttons, *b2* and *b3*, to the procedural implementation so as to conform to the following illustration (note that in SetDate state, pressing *b3* results with increasing the current date by 1):



Note that in SetDate state, pressing *b3* results with increasing the minute with 1.

Upon completion of the change, write a small test to ensure that whenever a “button is pressed” the appropriate action is taken, for example by printing a message on the current time and current state to the standard input.

**(Task 9)** Add a state to the procedural implementation: Add SetHour, SetSeconds states to the procedural implementation so as to conform to the following illustration:



Note that in SetHour (SetSeconds) state, pressing *b3* results with increasing the minutes (seconds) with 1.

Upon completion of the change, write a small test to ensure that whenever a “button is pressed” the appropriate action is taken, for example by printing a message on the current time and current state to the standard input.



**Table 9.** Summary sheet

Upon completion of all tasks, summarize the total duration of all tasks in this table. Add any comments you find relevant (increase table size if necessary.)

<b>TASK</b>	<b>TOTAL LENGTH</b>	<b>COMMENTS</b>
1.		
2.		
3.		
4.		
5.		
6.		
7.		
8.		
9.		