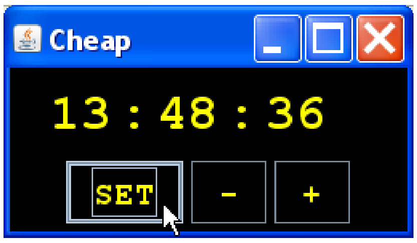
Cheap Digital Clock

Assignment

Write a graphical application for a *cheap* digital alarm clock. The clock has 3 buttons, an LED display for the time, an LED or icon to indicate if the alarm is on, but no others controls -- controls cost money and this is a *cheap* clock.

The buttons on the clock are to set the alarm time, turn the alarm on/off, or silence the alarm when it rings.



What to Submit

1. Commit source code of your application to BitBucket as pa5. In your repository you should have folders named `src` (source code) and `dist` (distribution). Please don't create an extra layer of folders, like pa5/PA5. But you *should* use packages inside the src/ folder.

2. Include a *runnable* JAR file of your application. It's size must be less than or equal 1MB. Commit it as **`pa5/dist/DigitalClock.jar`**

3. Submit on **paper** a **UML class diagram** of your design. In the diagram, omit simple get/set methods unless they have important meaning (like setting a state). The purpose of the UML diagram is to show structure and relationships between classes.

Evaluation

1. The clock implements the functional requirements and adheres to design requirements.
2. Application uses the *State Design Pattern* instead of "if" or "switch" to handle events like button press.
3. Good separation between the UI (view), controller, and clock (model).
4. Code quality: code follows coding and documentation guidelines for the course.

Functional Requirements

1. A digital clock with a graphical interface that shows the time in hours, minutes, and seconds.
2. Display is updated every second.
3. The clock has an **alarm** that the user can set and turn on/off.
4. The clock must be built *cheaply*, therefore it can contain **only**
   - display for hours : minutes : seconds
   - one LED or icon to indicate if the Alarm is on
   - 3 buttons for controlling the clock
   - No other components (no dialog boxes, pop-up help, etc)
5. *Visual feedback:* When setting the alarm time, the LED should blink or change color to indicate which field is being set.
6. When the alarm occurs, play a sound and blink the display. Don't use big audio files or try to download music from the Internet. A small MP3, MIDI, or WAV clip is OK.

7. Application should **exit** when window is closed. Don't forget jframe.setDefaultCloseOperation(...). You can add ability to show time in tray icon when clock window is minimized, but its not required.

Design Requirements

1. Separate the UI (View) from the logic (Model) of the digital clock. You *may* also have a controller class.

2. Use the *State Pattern* for handling actions requested by the UI. The state pattern can also be used to decide what to display (current time, setting alarm time, sounding alarm).

Programming Hints

1. The actions performed by the buttons depends on the *state* of the clock.

Display Time Mode:
"set" button changes the state to *Setting Alarm mode*,
"plus" button displays the alarm time (so user can check it). You can either toggle between *Display Time* and *Display Alarm Time* or use press-and-hold to display alarm time.

Setting Alarm Mode:
"set" button causes mode to change which field is being set: hours → minutes → seconds → exit.
"plus" and "minus" buttons increase or decrease the value of the current field (alarm hours, alarm minutes, alarm seconds).
You can treat each of these as a separate state, e.g. SetAlarmHours, SetAlarmMinutes, … There behavior is similar so you might treat them as substates or write a common superclass for them.

Alarm Ringing Mode: when the alarm is ringing, any button press turns the alarm off.

2. Setting alarm time: If user increments (say) minutes 58 → 59 → 0, ***don't*** increment the hours field when the minutes wrap to 0. This is for consistency with other clocks. *Familiarity* is good UI design.

3. Don't Set the Time. Digital Clock always uses the time from the operating system.

How to Update the Display Time

Use a timer running as a separate thread. There are many ways to do it; here are two.

1. Use a java.util.Timer and a TimerTask. The Timer will call your TimerTask according to a schedule, and TimerTask will tell Clock to update the time. The Timer is providing the timing loop, so your run method does not need a loop or sleep.

```
public class ClockTask extends TimerTask {
     public void run( ) {
          clock.updateTime( );
     }
}
// From somewhere in your application, run:
    final static long INTERVAL = 500; // millisec
    public void main( String [] args ) {
        Clock clock = new Clock( );
        //... add observers and then:
        TimerTask clocktask = new ClockTask( clock );
        Timer timer = new Timer();
        long delay = 1000 - System.currentTimeMills()%1000;
        // you can also use timer.schedule here
        timer.scheduleAtFixedRate( clocktask, delay, INTERVAL );
    }
```

2. Use a javax.swing.Timer. This Timer will notify ActionListeners at fixed intervals.

Swing Timer has a benefit for graphical apps: the ActionListener is always invoked in the *Event Dispatcher Thread*, which is the thread that is *supposed* to update UI components.

```
int delay = 500; // milliseconds
ActionListener task = new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        clock.updateTime();
    }
};
javax.swing.Timer timer = new javax.swing.Timer(delay, task);
timer.start();
```

Don't Use Runnable and Thread

Its possible to implement a timer using a *Runnable* object that's launched in a *Thread*. This method is not recommended because it is less accurate and a poor abstraction. The two Timer classes mentioned above have methods that automatically correct for inaccuracies in timer interval.

Design the States

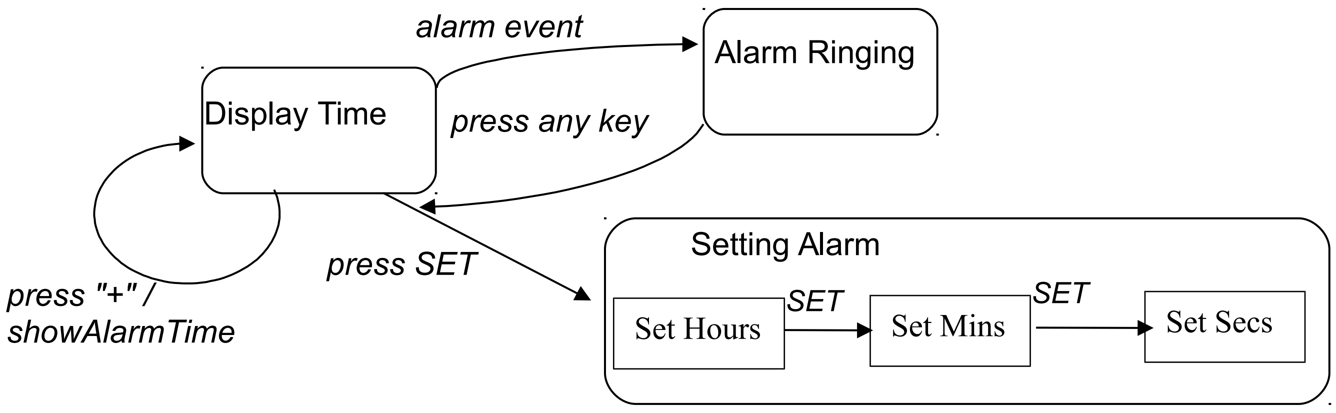
You should invent your own state machine design, but here is an example

"Display Time" - display the current time

"Setting Alarm" with sub-states "Set Hours", "Set Minutes", "Set Seconds", and "Alarm on/off".

"Alarm Ringing"

The buttons behave differently in each state and the display is updated differently in each state.



You can *simplify* your clock by writing one class for each state, as in the *State Pattern*. The UI sends button events to a controller, and the controller sends the button event to the current state.

Each state object knows how to handle button and timer events for its own state only, which makes the code simpler (almost no "if" statements). When a state wants to change to another state, it tells the controller to change state.

Object-Oriented Approach to State Pattern

The O-O approach is use an *object* for each state and *delegate* behavior to that object -- like the Strategy Pattern, with one Strategy for each state.

1. Determine *what behavior* (methods) depends on states.

2. Write an *interface* or *abstract superclass* for these methods. An *abstract superclass* is useful if you have some common behavior you want states to inherit, so you don't have to write it in many state classes. You can use the superclass to keep a reference to the clock or controller class.

3. In the Clock class, *delegate* the methods to the state objects.

Example

Suppose we determine that the clock responds to "SetKeyPressed" and "updateTime" differently in each state. We define an abstract class for the states that includes these methods:

```
public abstract class ClockState {
    protected Clock clock; // clock model or controller
    public ClockState(Clock clock) { this.clock = clock; }
    /** handle "set" key press */
    public abstract void performSet( ) ;
    /** handle updateTime event notification */
    public abstract void updateTime();
    /** something to do before exiting this state */
    public void leaveState( ) { };
    //TODO What other methods depend on state?
    //     Don't just copy this code
}

/** a Clock State for Displaying Time */
public class DisplayTimeState extends ClockState {
    public DisplayTimeState(Clock clock) { super(clock); }
    /** set key changes to the SetAlarm state */
    public void performSet( ) {
     clock.setstate( clock.SET_ALARM_STATE );
    }
    public void updateTime() { /* update the ui */ }
}
public class Clock {
    private ClockState state;
    .
    .
    .
    /** delegate behavior to state objects */
    public void updateTime( ) { state.updateTime(); }
    public void handleSetKey()  { state.performSet(); }
}
```

Enter State and Leave State

In the State Pattern it is often useful to define two special methods: enterState() and leaveState(). These methods give the states a chance to perform a special activity whenever the state is entered or before it leaves. For example, turn off the alarm or stop blinking.

In the superclass, these methods are *not* abstract so subclasses don't have to implement them.

```
public abstract class ClockState {
   /** this method is called by setState when we exit this state */
   public void leaveState() { /* default does nothing */ }
```

Accessing Files that are in a JAR file

Your Clock will be run as a JAR file, so any image files, sound files, and special fonts must be included in the JAR file. To load images and sounds, you can't use "new FileInputStream(*filename*)".

When you create the JAR file, everything from your "bin" directory (created by Eclipse from files and folders in your "src" directory) is included in the JAR file and added to the classpath. Even image files and sound files.

The trick is to use a **`Class`** or **`ClassLoader`** object to load images and sounds from the classpath (that is, from the JAR file). **`Class`** and **`ClassLoader`** both have methods **`getResource`**(*String name*) and **`getResourceAsStream`**(*String name*) that can find a file anywhere on the classpath. The contents of your JAR file will be on the classpath.

In your project, put resources in a separate folder that is inside your "src" folder. For example, an `images` directory (or "`resource`" directory). Suppose you have this:

```
src/images/alarm.gif
```

This directory is part of your source code tree so that it will be included in the JAR file.

To get a URL for `alarm.gif` use:

```
Class clazz = this.getClass();

URL alarm = clazz.getResource("images/alarm.gif");

ImageIcon icon = new ImageIcon( alarm );
```

To create an InputStream for reading a resource file, use

```
InputStream in = clazz.getResourceAsStream("images/alarm.gif")
```

You can also load your own *Font* from a file using a resource.

Don't Create Lots of Image or Sound Objects

Creating a lot of images wastes memory and CPU time. Don't create a new image each time you need to update the display.

You should only create **one instance** of each image or sound your application needs. Save them as (final) attributes, an enumeration, a HashMap, or some other attribute so you can reuse them in your UI.

You can use the *same* image object in several GUI components. For example, the same Image object of "0" can be used 5 times to display "20:00:00".

Avoid Garbage: Don't Create a Lot of String or Date Objects

Your application needs to update the time every second. **Don't do this:**

```
long now = System.currentTimeMillis( );
String time = String.format("%tT", now);
label.setText( time );
```

This (bad) code **creates a new String object every second!** A String contains a **`char`** array, so you are creating **two objects every second**. Update the time **without creating new objects**. You can use just *one* Date object (defined as an attribute) and update it with the current time each second. Then use the Date to get hours, minutes, and seconds:

```
// attribute
private Date time = new Date();
public void updateTime( ) {
     time.setTime( System.currentTimeMillis() );
     int hours = time.getHours();
     int minutes = time.getMinutes();
```

There are lots of alternatives to Date. The key point is to either use primitives like int or a *mutable* object that you can update.

Similarly, if you use JLabel objects for each digit on the clock, and set the ImageIcon in the JLabel (so you get a nice, *cheap looking* display) for each digit on the clock display, then you should create an array of 10 ImageIcon objects for digits, and use jlabel.setIcon(digits[k]) to update the icons without creating new icon objects each time.

Requirement for Individual Work

1. All submitted work must be your own. No group work.
2. You can discuss ideas, share UML diagrams, and share sample code (like you'd find on StackOverflow), but not share application code. You can ask the TAs for help on anything, but don't ask them to design the application for you.
3. Anyone involved in copying will fail the course and be reported to the university.

Resources

http://kan-chan.stbbs.net/download/digits/maine.html - LED digits as GIF or PNG. For cheap looking, LED style digits.

Head First Design Patterns chapter 10 covers the *State Pattern* and is easy to read. (Available on se.cpe.ku.ac.th)

Wikipedia, http://en.wikipedia.org/wiki/UML_state_machine examples of how to use a State Machine.

Wikipedia, http://en.wikipedia.org/wiki/State_diagram how to read a State Diagram.