1. Design Patterns
    a. Mediator
    b. Singleton
    c. Strategy
    d. Adapter
    e. Template Method
2. State Design Pattern

```
class Swamp
{
    user_wants_access_to_swamp() {
            attempt_login()
    }

    attempt_login() {
            load_login_page()
            if (new_user) {
                    register_new_user()
            }
            else { // existing_user
                    login_existing_user()
            }
    }

    register_new_user() {
            load_new_user_page()
            get_new_user_information()
            complete_registration()
            load_home()
    }

    login_existing_user() {
            load_existing_user_page()
            get_existing_user_information()
            authenticate()
            load_home()
    }

    load_home() {
            load_home_page()
            if (email_not_verified) {
                    attempt_login()
            }
            else if (user_selects_product) {
                    load_cart()
            }
            else if (user_clicks_logout) {
```

```
                logout()
        }
        else {
                ...
        }
    }

    load_cart() {
        load_cart_page()
        if (user_wants_add_more_items) {
                load_home()
        }
        else if (user_clicks_logout) {
                logout()
        }
        else if (user_ready_to_pay) {
                checkout()
        }
        else {
                ...
        }
    }

    checkout() {
        load_checkout_page()
        is_successful = charge_user()
        if (not is_successful) { // payment declined
                load_cart()
        }

        if (user_clicks_logout) {
                logout()
        }
    }

    logout() {
        log_user_out()
        load_logout_page()
    }
}
```

3. Design Pattern Application – Strategy

```
class PresidentialPolls
{
    GraphBehavior gb;

    PresidentailPolls(GraphBehvaior gb) {
        this.gb = gb;
    }

    drawGraph() {
        gb.drawGraph()
    }

    setGraphBehavior(GraphBehavior gb) {
        this.gb = gb;
    }
}

interface GraphBehavior {
    drawGraph()
}

class BarGraphBehavior implements GraphBehavior {
    drawGraph() {
        // draw bar graph implementation
    }
}

class PieGraphBehavior implements GraphBehavior {
    drawGraph() {
        // draw pie graph implementation
    }
}
```

4. Sequence Diagram

```
class Actor
{
    SecureLogin sl;

    use_service() {
        sl.requestLogin()
        ...
    }
}

class SecureLogin
{
    string userType;
    AccountDB accDB;

    displayLoginScreen() {
        ...
    }

    displayAdmin() {
        ...
    }

    displayUser() {
        ...
    }

    requestLogin() {
        displayLoginScreen()
        userType = "invalid"
        while (userType == "invalid") {
            userType = accDB.isValid(username, password)
            if (userType == "admin") {
                displayAdmin()
            }
            else if (userType == "user") {
                displayUser()
            }
            else if (userType == "invalid") {
                displayLoginScreen()
            }
            else {
                ...
            }
        }
```

```
        }
    }

    class AccountDB
    {
        System sys;

        isValid(username, password) {
                userType = sys.isInDatabase(username)
                return userType
        }
    }

    class System
    {
        isInDatabase(username) {
                userType = get_usertype_by_username()
                return userType;
        }
    }
```

5. Design Pattern in Practice
    a. Concrete strategy classes – QuadTreeDrawing, ODGDrawing, DefaultDrawing
    b. drawing.setFontRenderContext(g.getFontRenderContext());
       drawing.drawCanvas(g);

       The Information Hiding Principle tells us that we should create stable interfaces for features that are unlikely to change. The parts that are likely to change should be encapsulated and hidden away, so that when they are changed, there are fewer side effects.

       DefaultDrawingView.drawCanvas() will not have to be modified if a new concrete strategy class of the strategy interface Drawing is desired. The programmer will have to create a new concrete strategy class that implements the Drawing interface. Then, the user simply has to set the private drawing variable to be an instance of the new concrete strategy class.

       In this way, it is fairly easy to program new concrete Drawing strategy classes as the functions that use the strategy class should not have to be changed because of the design pattern.
    c. Differences from traditional Singleton
        i. ClipboardUtil provides a function setClipboard(Clipboard instance) { ClipboardUtil.instance = instance; } This function can be invoked with null so that the instance becomes null. Calling getClipboard() will then create a new instance. In the traditional way, once the Singleton instance is created, a new one is never created.
        ii. In the traditional Singleton, the instance stored in the Singleton class can only be an instanceof one class. In ClipboardUtil, the Clipboard can be an instanceof OSXClipboard, JNLPClipboard, or AWTClipbaord.
        iii. In the traditional Singleton, the instance stored is typically of the Singleton type. The ClipboardUtil Singleton class stores a Clipboard instance, not a ClipboardUtil instance.
        iv. In the traditional Singleton, the creation of the Singleton instance is synchronized to be thread-safe. It does not seem like ClipboardUtil provides any mechanisms to be thread-safe (with synchronized keyword or by other means).