Smart Password Strength Checker

A Java application for real-time password analysis and feedback.

Xinyuan Fan (Amber) & Xujing Hui

2025 August

Project Overview

Our Smart Password Strength Checker is a Java desktop application designed to provide instant, actionable feedback on password strength.

Real-time Feedback

Users receive red (weak), orange (medium), or green (strong) indicators as they type, along with specific suggestions for improvement.

Enhanced Security

In a world of increasing cyber threats, our tool simplifies understanding password strength, helping users create robust defenses against unauthorized access.

Solid Architecture

Built with the Model-View-Controller (MVC) pattern and adhering to key OOP principles (encapsulation, inheritance, polymorphism, abstraction). Rigorously tested with 51 passing test cases.

Technical Architecture: MVC Pattern

We chose the Model-View-Controller (MVC) design pattern for its clear separation of concerns, modularity, testability, and maintainability.

Model

Contains business logic: PasswordAnalyzer and validation rules (LengthRule, CharacterVarietyRule, CommonPatternRule). Extensible via interfaces.

View

Handles UI elements: Java Swing GUI provides real-time, intuitive feedback to users.

Controller

Coordinates Model and View: Processes user input, calls Model for analysis, and updates View with results.

Code Walkthrough: Key Implementations

Let's dive into the core components that power our Smart Password Strength Checker, focusing on the foundational interfaces and rule implementations.

PasswordRule Interface

Defines a contract for all validation rules, emphasizing **abstraction**. Each rule must implement `validate()` and `getFeedback()`.

```
public interface PasswordRule {
  boolean validate(String password);
  String getFeedback(String password);
}
```

LengthRule Implementation

An example of a concrete rule, demonstrating **encapsulation** with a private minimum length field. It validates password length and provides specific feedback.

```
public class LengthRule implements PasswordRule {
  private final int minLength;
  // ... constructor and methods ...
}
```

CharacterVarietyRule

A more complex rule checking for uppercase, lowercase, digits, and special characters. This showcases **polymorphism**, as different rules share the same interface methods.

```
public class CharacterVarietyRule implements PasswordRule {
  @Override
  public boolean validate(String password) {
    // ... logic for character types ...
  }
  // ... getFeedback method ...
}
```

Code Walkthrough: Model (PasswordAnalyzer)

The `PasswordAnalyzer` is the heart of our Model, applying validation rules to determine password strength. It leverages **polymorphism** by interacting with rules through the `PasswordRule` interface, ensuring flexibility and extensibility for future rule additions.

```
public class PasswordAnalyzer {
    private final List<PasswordRule> rules;

    public PasswordAnalyzer() {
        rules = new ArrayList<>();
        rules.add(new LengthRule(8));
        rules.add(new CharacterVarietyRule());
        rules.add(new CommonPatternRule());
    }

    public PasswordStrengthFeedback analyze(String password) {
        List<String> feedback = new ArrayList<>();
        int passed = 0;

        for (PasswordRule rule : rules) { // Polymorphism in action
            if (rule.validate(password)) {
                passed++;
            } else {
                String fb = rule.getFeedback(password);
                if (fb != null) feedback.add(fb);
            }
        }

        String strength;
        if (passed == rules.size()) strength = "Strong";
        else if (passed >= rules.size() - 1) strength = "Medium";
        else strength = "Weak";

        return new PasswordStrengthFeedback(strength, feedback);
    }
}
```

Dynamic Rule Application

The analyzer iterates through a list of `PasswordRule` objects, executing their validation logic without needing to know their specific types.

Extensible Design

Adding new password validation rules is simple: just implement the `PasswordRule` interface and add the new rule to the analyzer's list.

Strength Scoring

Based on the number of passed rules, the analyzer assigns a strength rating (Weak, Medium, Strong) and compiles specific feedback messages for the user.

Code Walkthrough: Controller

Controller Component

PasswordStrengthController: Coordinates between Model and View

```
public void onPasswordChanged(String password) {
    if (gui != null) {
        // Call Model for processing
        PasswordStrengthFeedback result = analyzer.analyze(password);

        // Update View with results
        gui.updateStrength(result.getStrength());
        gui.updateFeedback(result.getFeedback());
    }
}
```

Receives user input from View

Calls Model for processing

Updates View with results

Code Walkthrough: View

GUI(Interface)

Interface for GUI components

```
public interface GUI {
    void updateStrength(String strength);
    void updateFeedback(List<String> feedback);
}
```

Enables mock testing

Loose coupling

Testability

The View handles all user interface elements. When the user types, it delegates the action to the Controller. The GUI implements an interface, making it testable without the actual Swing components.

Code Walkthrough: View

CheckerGUI (View)

User interface implementation

```
// Listen for user typing
passwordField.addKeyListener(new KeyAdapter() {
    public void keyReleased(KeyEvent e) {
        controller.onPasswordChanged(passwordField.getText());
    }
});


// Update strength color
public void updateStrength(String strength) {
    switch (strength) {
        case "Strong": setColor(GREEN); break;
        case "Medium": setColor(ORANGE); break;
        default: setColor(RED);
    }
}
```

Java Swing GUI

Real-time updates

Color-coded feedback.

Benefits of MVC Architecture

The Model-View-Controller (MVC) pattern provides significant advantages, ensuring our Smart Password Strength Checker is robust, scalable, and easy to manage.

Separation of Concerns

Each component (Model, View, Controller) has a single, well-defined responsibility, simplifying development and debugging. For example, the GUI solely handles display, not analysis logic.

Enhanced Testability

Individual components can be tested in isolation, such as validating the Controller's logic using a mock GUI, which streamlines unit testing and ensures reliability.

Improved Maintainability

Changes or updates in one part of the application, like the user interface, minimally impact other parts, reducing the risk of introducing new bugs and simplifying ongoing support.

Greater Extensibility

Adding new features, such as additional password validation rules, primarily requires updates within the Model, allowing for easy expansion without disrupting the core architecture.

OOP Principles Demonstrated

Encapsulation

Private fields with public getters

(e.g., PasswordStrengthFeedback) protect data and provide clean interfaces.

Inheritance

Rule classes implement the PasswordRule interface, inheriting contracts for specific implementations.

Polymorphism

PasswordAnalyzer works with any PasswordRule implementation, allowing flexible and extensible rule processing.

Abstraction

Interfaces (PasswordRule, GUI) hide complex implementation details, simplifying testing and future development.

Comprehensive Testing Strategy

Our application is backed by a robust testing strategy, ensuring reliability and confidence in our code.

Test Coverage

- Unit Tests: Individual component validation.
- Integration Tests: Full MVC flow verification.
- Performance Tests: Scalability and efficiency checks.
- Edge Cases: Robustness against null/empty inputs.

Results

- 51 test methods executed.
- 100% pass rate achieved.
- Confidence in code correctness and maintainability.

Example Test:

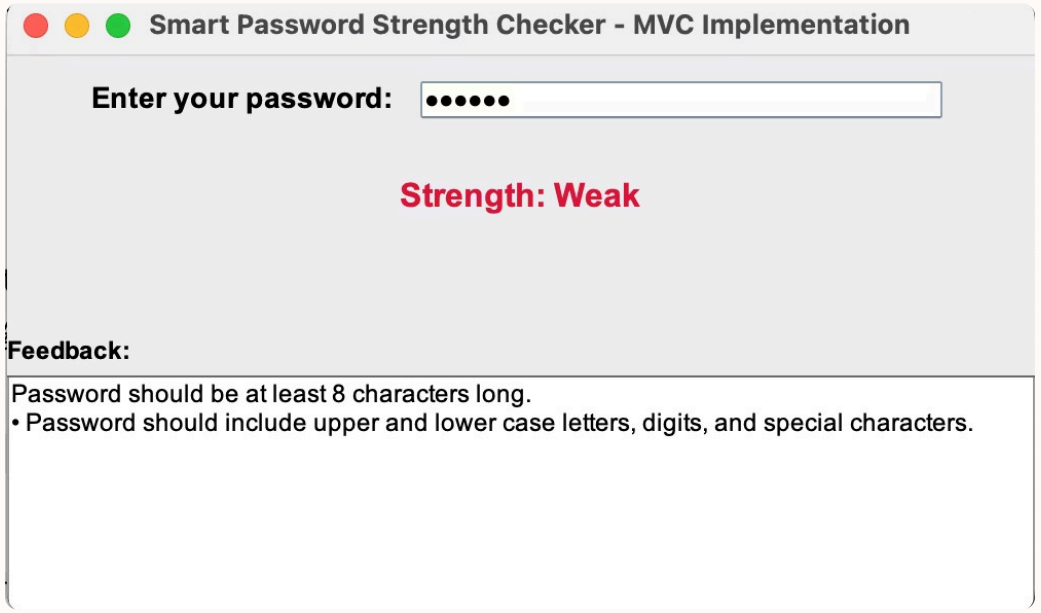
```
@Test void testStrongPassword() {
PasswordStrengthFeedback feedback =
analyzer.analyze("Str0ng!Passw0rd");
assertEquals("Strong", feedback.getStrength());
assertTrue(feedback.getFeedback().isEmpty()); }
```

Live Demonstration

Experience the Smart Password Strength Checker in action, showcasing its real-time analysis and intuitive feedback.

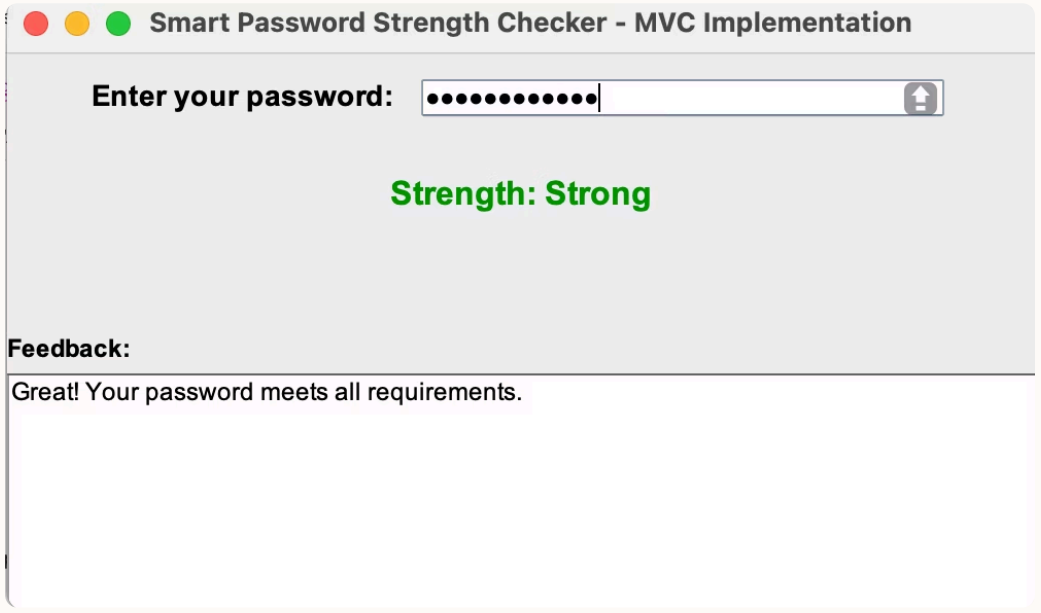
Weak Password

Typing "279304726 shows "Weak" in red, with suggestions for length, variety, and common patterns.



Strong Password

Typing "Abc37492749!" results in "Strong" in green, confirming all criteria are met.



Real-time Updates: Feedback and color-coding update instantly as you type, providing immediate guidance.

Challenges and Solutions

MVC Implementation

Challenge: Ensuring proper separation of Model, View, and Controller.

Solution: Clear interfaces and defined communication channels.

GUI Testing

Challenge: Difficulty in testing GUI components directly.

Solution: GUI interface for mock object testing of controller logic.

Rule Extensibility

Challenge: Making it easy to add new validation rules.

Solution: Interface design allows simple implementation and addition of new rules.

Performance

Challenge: Ensuring responsiveness with complex validation.

Solution: Efficient algorithms and data structures for fast analysis.

Future Enhancements

Our project lays a strong foundation for future development, expanding its capabilities and user experience.

Enhanced Security

- Integrate with compromised password databases.
- More sophisticated pattern detection.

User Experience

- Password history tracking.
- Export functionality.
- Multi-language support.

Advanced Features

- Password generation and strength comparison tools.
- Integration with password managers.

Scalability

- Enterprise features: user management, reporting.

Conclusion

The Smart Password Strength Checker demonstrates professional software engineering practices and provides real value.

Key Achievements

Fully functional, well-tested application with real user value.

Learning Outcomes

Deepened understanding of OOP, design patterns, testing, and GUI development.

Technical Excellence

Modular, maintainable, extensible code with MVC architecture and comprehensive testing.

Real-World Application

Translates theoretical concepts into practical, deployable software.

Q&A

We're happy to answer any questions about our implementation, design decisions, or technical details.