For Equivalence Class Testing, we can partition the inputs into valid (positive) and invalid (negative) groups based on their respective equivalence classes for the automated banking application requirements. Here's how these classes break down based on your document:

Equivalence Classes

1. **Area Code**:
   - Valid: A blank or three-digit number (000-999).
   - Invalid: Anything outside of a three-digit range, such as non-numeric characters or values above 999.
2. **Prefix**:
   - Valid: A three-digit number between 200 and 999.
   - Invalid: Numbers less than 200, greater than 999, or any non-numeric inputs.
3. **Suffix**:
   - Valid: A four-digit number (1000-9999).
   - Invalid: Numbers outside of this range or any non-numeric inputs.
4. **Password**:
   - Valid: Six alphanumeric characters.
   - Invalid: Any password that is not exactly six characters, or contains special characters outside alphanumeric scope.
5. **Command**:
   - Valid: "Check status", "Deposit", "Withdrawal".
   - Invalid: Any command other than these three.

```
import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;


public class BankingApplicationEquivalenceTest {


    // Helper method to simulate validation

    public boolean validateInput(String areaCode, String prefix, String suffix, String password, String command) {

        return (areaCode.isEmpty() || areaCode.matches("\\d{3}")) &&

            prefix.matches("\\d{3}") && Integer.parseInt(prefix) >= 200 && Integer.parseInt(prefix) <= 999 &&

            suffix.matches("\\d{4}") && Integer.parseInt(suffix) >= 1000 && Integer.parseInt(suffix) <= 9999 &&

            password.matches("\\w{6}") &&

            (command.equals("Check status") || command.equals("Deposit") || command.equals("Withdrawal"));

    }
```

```
// Positive Test Cases
@Test
public void testValidEmptyAreaCode() {
    assertTrue(validateInput("", "200", "1000", "abc123", "Check status"));
}

@Test
public void testValidThreeDigitAreaCode() {
    assertTrue(validateInput("123", "300", "2000", "def456", "Deposit"));
}

@Test
public void testValidMinPrefix() {
    assertTrue(validateInput("456", "200", "3000", "xyz789", "Withdrawal"));
}

@Test
public void testValidMaxPrefix() {
    assertTrue(validateInput("789", "999", "4000", "abcd12", "Check status"));
}

@Test
public void testValidSuffix() {
    assertTrue(validateInput("555", "600", "5000", "lmno34", "Deposit"));
}

@Test
public void testValidMinPassword() {
    assertTrue(validateInput("111", "800", "6000", "efgh56", "Withdrawal"));
}

@Test
```

```
public void testValidMaxPassword() {
    assertTrue(validateInput("222", "250", "7000", "ijkl78", "Check status"));
}

@Test
public void testValidCommandCheckStatus() {
    assertTrue(validateInput("333", "700", "8000", "pass34", "Check status"));
}

@Test
public void testValidCommandDeposit() {
    assertTrue(validateInput("444", "350", "9000", "mnop90", "Deposit"));
}

@Test
public void testValidCommandWithdrawal() {
    assertTrue(validateInput("555", "450", "9999", "qrstu1", "Withdrawal"));
}

// Negative Test Cases
@Test
public void testInvalidAreaCodeNonNumeric() {
    assertFalse(validateInput("ABC", "200", "1000", "abc123", "Check status"));
}

@Test
public void testInvalidAreaCodeAboveRange() {
    assertFalse(validateInput("1000", "250", "2000", "def456", "Deposit"));
}

@Test
public void testInvalidPrefixBelowRange() {
```

```
    assertFalse(validateInput("666", "199", "3000", "ghi789", "Withdrawal"));
}

@Test
public void testInvalidPrefixAboveRange() {
    assertFalse(validateInput("777", "1000", "4000", "jkl012", "Check status"));
}

@Test
public void testInvalidSuffixBelowRange() {
    assertFalse(validateInput("888", "300", "0999", "mno345", "Deposit"));
}

@Test
public void testInvalidSuffixAboveRange() {
    assertFalse(validateInput("999", "400", "10000", "pqr678", "Withdrawal"));
}

@Test
public void testInvalidPasswordTooShort() {
    assertFalse(validateInput("111", "500", "2000", "abcd1", "Check status"));
}

@Test
public void testInvalidPasswordTooLong() {
    assertFalse(validateInput("222", "600", "3000", "abcdef1", "Deposit"));
}

@Test
public void testInvalidCommandNotRecognized() {
    assertFalse(validateInput("333", "700", "4000", "xyz456", "Transfer"));
}
```

```
    @Test
    public void testInvalidPrefixNonNumeric() {
        assertFalse(validateInput("444", "ABC", "5000", "uvw890", "Withdrawal"));
    }
}
```

Explanation

1. **Positive Test Cases**: These tests cover the valid equivalence classes by providing inputs within the accepted ranges and formats.
2. **Negative Test Cases**: These tests cover invalid equivalence classes with inputs that are out of range or do not meet format requirements.
3. **JUnit Assertions**: `assertTrue` confirms expected valid results, while `assertFalse` checks expected invalid cases.
4. **validateInput Method**: This helper method performs input validation according to the specifications for area code, prefix, suffix, password, and command.

Run these tests in Eclipse Oxygen using JUnit to verify the equivalence class coverage for your banking application.