OBJ02-J. Preserve dependencies in subclasses when changing superclasses

Developers often separate program logic across multiple classes or files to modularize code and to increase reusability. When developers modify a superclass (during maintenance, for example), the developer must ensure that changes in superclasses preserve all the program invariants on which the subclasses depend. Failure to maintain all relevant invariants can cause security vulnerabilities.

Noncompliant Code Example

In this code example, a class `Account` stores banking-related information without any inherent security. Security is delegated to the subclass `BankAccoun` . The client application is required to use `BankAccount` because it contains the security mechanism.

```
class Account {
  // Maintains all banking-related data such as account balance
  private double balance = 100;

  boolean withdraw(double amount) {
    if ((balance - amount) >= 0) {
      balance -= amount;
      System.out.println("Withdrawal successful. The balance is : "
                         + balance);
      return true;
    }
    return false;
  }
}

public class BankAccount extends Account {
  // Subclass handles authentication
  @Override boolean withdraw(double amount) {
    if (!securityCheck()) {
      throw new IllegalAccessException();
    }
    return super.withdraw(amount);
  }

  private final boolean securityCheck() {
    // Check that account management may proceed
  }
}

public class Client {
  public static void main(String[] args) {
    Account account = new BankAccount();
    // Enforce security manager check
    boolean result = account.withdraw(200.0);
    System.out.println("Withdrawal successful? " + result);
  }
}
```

At a later date, the maintainer of the `Account` class added a new method called `overdraft()`. However, the `BankAccount` class maintainer was unaware of the change. Consequently, the client application became vulnerable to malicious invocations. For example, the `overdraft()` method could be invoked directly on a `BankAccount` object, avoiding the security checks that should have been present. The following noncompliant code example shows this vulnerability:

```
class Account {
  // Maintains all banking-related data such as account balance
  private double balance = 100;

  boolean overdraft() {
    balance += 300;     // Add 300 in case there is an overdraft
    System.out.println("Added back-up amount. The balance is :"
                       + balance);
    return true;
  }

  // Other Account methods
}

public class BankAccount extends Account {
  // Subclass handles authentication
  // NOTE: unchanged from previous version
  // NOTE: lacks override of overdraft method
}

public class Client {
  public static void main(String[] args) {
    Account account = new BankAccount();
    // Enforce security manager check
    boolean result = account.withdraw(200.0);
    if (!result) {
      result = account.overdraft();
    }
    System.out.println("Withdrawal successful? " + result);
  }
}
```

Although this code works as expected, it adds a dangerous attack vector. Because the `overdraft()` method has no security check, a malicious client can invoke it without authentication:

```
public class MaliciousClient {
  public static void main(String[] args) {
    Account account = new BankAccount();
    // No security check performed
    boolean result = account.overdraft();
    System.out.println("Withdrawal successful? " + result);
  }
}
```

Compliant Solution

In this compliant solution, the `BankAccount` class provides an overriding version of the `overdraft()` method that immediately fails, preventing misuse of the overdraft feature. All other aspects of the compliant solution remain unchanged.

```
class BankAccount extends Account {
  // ...
  @Override boolean overdraft() { // Override
    throw new IllegalAccessException();
  }
}
```

Alternatively, when the intended design permits the new method in the parent class to be invoked directly from a subclass without overriding, install a security manager check directly in the new method.

Noncompliant Code Example (`Calendar`)

This noncompliant code example overrides the methods `after()` and `compareTo()` of the class `java.util.Calendar`. The `Calendar.after()` method returns a `boolean` value that indicates whether or not the `Calendar` represents a time *after* that represented by the specified `Object` parameter. The programmer wishes to extend this functionality so that the `after()` method returns `true` even when the two objects represent the same date. The programmer also overrides the method `compareTo()` to provide a "comparisons by day" option to clients (for example, comparing today's date with the first day of the week, which differs among countries, to check whether it is a weekday).

```
class CalendarSubclass extends Calendar {
  @Override public boolean after(Object when) {
    // Correctly calls Calendar.compareTo()
    if (when instanceof Calendar &&
        super.compareTo((Calendar) when) == 0) {
      return true;
    }
    return super.after(when);
  }

  @Override public int compareTo(Calendar anotherCalendar) {
    return compareDays(this.getFirstDayOfWeek(),
                       anotherCalendar.getFirstDayOfWeek());
  }

  private int compareDays(int currentFirstDayOfWeek,
                          int anotherFirstDayOfWeek) {
    return (currentFirstDayOfWeek > anotherFirstDayOfWeek) ? 1
           : (currentFirstDayOfWeek == anotherFirstDayOfWeek) ? 0 : -1;
  }

  public static void main(String[] args) {
    CalendarSubclass cs1 = new CalendarSubclass();
    cs1.setTime(new Date());
    // Date of last Sunday (before now)
    cs1.set(Calendar.DAY_OF_WEEK, Calendar.SUNDAY);
    // Wed Dec 31 19:00:00 EST 1969
    CalendarSubclass cs2 = new CalendarSubclass();
    // Expected to print true
    System.out.println(cs1.after(cs2));
  }

  // Implementation of other Calendar abstract methods
}
```

The `java.util.Calendar` class provides a `compareTo()` method and an `after()` method. The `after()` method is documented in the *Java API Reference* [API 2014] as follows:

> *The `after()` method returns whether this `Calendar` represents a time after the time represented by the specified `Object`. This method is equivalent to*
> *`compareTo(when) > 0`*
> *if and only if when is a `Calendar` instance. Otherwise, the method returns `false`.*

The documentation fails to state whether `after()` invokes `compareTo()` or whether `compareTo()` invokes `after()`. In the Oracle JDK 1.6 implementation, the source code for `after()` is as follows:

```
public boolean after(Object when) {
  return when instanceof Calendar
         && compareTo((Calendar) when) > 0;
}
```

In this case, the two objects are initially compared using the overriding `CalendarSubclass.after()` method, which invokes the superclass's `Calendar.after()` method to perform the remainder of the comparison. But the `Calendar.after()` method internally calls the `compareTo()` method, which delegates to `CalendarSubclass.compareTo()`. Consequently, `CalendarSubclass.after()` actually calls `CalendarSubclass.compareTo()` and returns `false`.

The developer of the subclass was unaware of the implementation details of `Calendar.after()` and incorrectly assumed that the superclass's `after()` method would invoke only the superclass's methods without invoking overriding methods from the subclass. MET05-J. Ensure that constructors do not call overridable methods describes similar programming errors.

Such errors generally occur because the developer made assumptions about the implementation-specific details of the superclass. Even when these assumptions are initially correct, implementation details of the superclass may change without warning.

Compliant Solution (`Calendar`)

This compliant solution uses a design pattern called Composition and Forwarding (sometimes also called Delegation) [Lieberman 1986], [Gamma 1995]. The compliant solution introduces a new *forwarder* class that contains a private member field of the `Calendar` type; this is *composition* rather than inheritance. In this example, the field refers to `CalendarImplementation`, a concrete instantiable implementation of the `abstract Calendar` class. The compliant solution also introduces a wrapper class called `CompositeCalendar` that provides the same overridden methods found in the `CalendarSubclass` from the preceding noncompliant code example.

```java
// The CalendarImplementation object is a concrete implementation
// of the abstract Calendar class
// Class ForwardingCalendar
public class ForwardingCalendar {
  private final CalendarImplementation c;

  public ForwardingCalendar(CalendarImplementation c) {
    this.c = c;
  }

  CalendarImplementation getCalendarImplementation() {
    return c;
  }

  public boolean after(Object when) {
    return c.after(when);
  }

  public int compareTo(Calendar anotherCalendar) {
    // CalendarImplementation.compareTo() will be called
    return c.compareTo(anotherCalendar);
  }
}

class CompositeCalendar extends ForwardingCalendar {
  public CompositeCalendar(CalendarImplementation ci) {
    super(ci);
  }

  @Override public boolean after(Object when) {
    // This will call the overridden version, i.e.
    // CompositeClass.compareTo();
    if (when instanceof Calendar &&
        super.compareTo((Calendar)when) == 0) {
      // Return true if it is the first day of week
      return true;
    }
    // No longer compares with first day of week;
    // uses default comparison with epoch
    return super.after(when);
  }

  @Override public int compareTo(Calendar anotherCalendar) {
    return compareDays(
             super.getCalendarImplementation().getFirstDayOfWeek(),
             anotherCalendar.getFirstDayOfWeek());
  }

  private int compareDays(int currentFirstDayOfWeek,
                          int anotherFirstDayOfWeek) {
    return (currentFirstDayOfWeek > anotherFirstDayOfWeek) ? 1
           : (currentFirstDayOfWeek == anotherFirstDayOfWeek) ? 0 : -1;
  }

  public static void main(String[] args) {
    CalendarImplementation ci1 = new CalendarImplementation();
    ci1.setTime(new Date());
    // Date of last Sunday (before now)
    ci1.set(Calendar.DAY_OF_WEEK, Calendar.SUNDAY);

    CalendarImplementation ci2 = new CalendarImplementation();
    CompositeCalendar c = new CompositeCalendar(ci1);
    // Expected to print true
    System.out.println(c.after(ci2));
  }
}
```

Note that each method of the class `ForwardingCalendar` redirects to methods of the contained `CalendarImplementation` class, from which it receives return values; this is the *forwarding* mechanism. The `ForwardingCalendar` class is largely independent of the implementation of the class `CalendarImplementation`. Consequently, future changes to `CalendarImplementation` are unlikely to break `ForwardingCalendar` and are also unlikely to break `CompositeCalendar`. Invocations of the overriding `after()` method of `CompositeCalendar` perform the necessary comparison by using the `CalendarImplementation.compareTo()` method as required. Using `super.after(when)` forwards to `ForwardingCalendar`, which invokes the `CalendarImplementation.after()` method as required. As a result, `java.util.Calendar.after()` invokes the `CalendarImplementation.compareTo()` method as required, resulting in the program correctly printing `true`.

Risk Assessment

Modifying a superclass without considering the effect on subclasses can introduce vulnerabilities. Subclasses that are developed with an incorrect understanding of the superclass implementation can be subject to erratic behavior, resulting in inconsistent data state and mismanaged control flow. Also, if the superclass implementation changes then the subclass may need to be redesigned to take into account these changes.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
OBJ02-J	Medium	Probable	High	**P4**	**L3**

Automated Detection

Sound automated detection is not currently feasible.

Related Vulnerabilities

The introduction of the `entrySet()` method in the `java.util.Hashtable` superclass in JDK 1.2 left the `java.security.Provider` subclass vulnerable to a security attack. The `Provider` class extends `java.util.Properties`, which in turn extends `Hashtable`. The `Provider` class maps a cryptographic algorithm name (for example, RSA) to a class that provides its implementation.

The `Provider` class inherits the `put()` and `remove()` methods from `Hashtable` and adds security manager checks to each. These checks ensure that malicious code cannot add or remove the mappings. When `entrySet()` was introduced, it became possible for untrusted code to remove the mappings from the `Hashtable` because `Provider` failed to override this method to provide the necessary security manager check [SCG 2009]. This situation is commonly known as the *fragile class hierarchy* problem.

Related Guidelines

Secure Coding Guidelines for Java SE, Version 5.0	Guideline 4-6 / EXTEND-6: Understand how a superclass can affect subclass behavior

Bibliography

[API 2014]	Class `Calendar`
[Bloch 2008]	Item 16, "Favor Composition over Inheritance"
[Gamma 1995]	*Design Patterns: Elements of Reusable Object-Oriented Software* (p. 20)
[Lieberman 1986]	"Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems"