

<http://www.developer.com/tech/article.php/3686731/Working-with-Design-Patterns-Composite.htm>

[Back to article](#)

Working with Design Patterns: Composite

July 2, 2007

A *composite* is an object that can contain other objects. The composite pattern can be used in situations where client code can treat composites in the same manner as non-composites. In this example of the composite design pattern, I demonstrate how the need to express complex conditionals in SQL can be neatly fulfilled using the composite pattern. Individual, or "primitive," clauses, such as `name like 'abc%'`, can be combined using conjunctions such as `and`. I can consider that an `and` expression is a composite clause that can consists of primitive clauses such as `like`.

To demonstrate how I can build up to a composite, I'll start by implementing my low-level needs. A `Column` object encapsulates the name of a column. Subclasses define any additional details required to construct various types of SQL statements. For example, the class `StringColumn` defines a `length` field that an SQL generator object might use to build a `create table` statement.

More importantly, the `Column` class defines an abstract method, `sqlValue`. Subclasses implement this method to return the value substring for an SQL where clause. For example, the SQL substring for the where clause where `name = 'abc'` would be `"'abc'"`. Numeric columns, on the other hand, don't get embedded in single quotes (tics). The SQL value for where `amount = 10` would be `"10"`.

```
public abstract class Column {
    private String name;

    public Column(String name) {
        this.name = name;
    }

    String getName() {
        return name;
    }

    abstract String sqlValue(Object value);
}
```

```
public class StringColumn extends Column {
    private int length;

    public StringColumn(String name, int length) {
        super(name);
        this.length = length;
    }

    public int getLength() {
        return length;
    }

    public String sqlValue(Object value) {
        return "'" + value + "'";
    }
}
```

```
public class NumericColumn extends Column {
    public NumericColumn(String name) {
        super(name);
    }

    public String sqlValue(Object value) {
        return value.toString();
    }
}
```

With the `Column` hierarchy in place, it's easy to write a test that demonstrates how to construct the where clause for a simple equality comparison.

```
import static org.junit.Assert.*;
import org.junit.*;

public class EqualsTest {
    @Test
    public void stringColumn() {
        Column column = new StringColumn("name", 10);
        Equals criteria = new Equals(column, "joe");
        assertEquals("name = 'joe'", criteria.sqlString());
    }

    @Test
    public void numericColumn() {
        Column column = new NumericColumn("amount");
        Equals criteria = new Equals(column, 5);
        assertEquals("amount = 5", criteria.sqlString());
    }
}
```

The implementation:

```
public class Equals {
    private Column column;
    private Object value;

    public Equals(Column column, Object value) {
        this.column = column;
        this.value = value;
    }

    public String sqlString() {
        return String.format("%s = %s",
            column.getName(), column.sqlValue(value));
    }
}
```

That was pretty simple!

I now want to write a where clause that combines two `EqualsCriteria`, producing an SQL statement such as:

```
name = 'Joe' and amount = 10
```

The class `AndTest` lays out an example:

```
import static org.junit.Assert.*;
import org.junit.*;

public class AndTest {
    @Test
```

```

    public void and() {
        Equals name = new Equals(new StringColumn("name", 1), "Joe");
        Equals amount = new Equals(new NumericColumn("amount", 5), 5);
        And and = new And(name, amount);
        assertEquals("name = 'Joe' and amount = 5", and.sql());
    }
}

```

Again, a very simple implementation:

```

public class And {
    private Equals left;
    private Equals right;

    public And(Equals left, Equals right) {
        this.left = left;
        this.right = right;
    }

    public String sql() {
        return left.sqlString() + " and " + right.sqlString();
    }
}

```

Moving on, I can start rapidly adding support for additional SQL where clause constructs. Like, for example, like:

```

// LikeTest.java:
import static org.junit.Assert.*;
import org.junit.*;

public class LikeTest {
    @Test
    public void simple() {
        Like like = new Like(new StringColumn("name", 1), "Joe%");
        assertEquals("name like 'Joe'", like.sql());
    }
}

// Like.java:
public class Like {
    private StringColumn column;
    private String value;

    public Like(StringColumn column, String value) {
        this.column = column;
        this.value = value;
    }

    public String sql() {
        return String.format("%s like %s",
            column.getName(), column.sqlValue(value));
    }
}

```

Right now, the And class supports joining only two Equals objects. I want to have And support Like objects as well, or any other conditional that I might dream up. I can extract a common interface that represents the ability of each to return an SQL representation:

```

public interface Criteria {
    String sqlString();
}

// EqualsTest.java:
import static org.junit.Assert.*;
import org.junit.*;

public class EqualsTest {
    @Test
    public void stringColumn() {
        Column column = new StringColumn("name", 10);
        Criteria criteria = new Equals(column, "joe");
        assertEquals("name = 'joe'", criteria.sqlString());
    }

    @Test
    public void numericColumn() {
        Column column = new NumericColumn("amount");
        Criteria criteria = new Equals(column, 5);
        assertEquals("amount = 5", criteria.sqlString());
    }
}

// Equals.java:
public class Equals implements Criteria {
    private Column column;
    private Object value;

    public Equals(Column column, Object value) {
        this.column = column;
        this.value = value;
    }

    public String sqlString() {
        return String.format("%s = %s",
            column.getName(), column.sqlValue(value));
    }
}

```

I can modify the Like class to implement the same Criteria interface. This requires me to do a bit of refactoring because I wasn't consistent with the method names.

```

// LikeTest.java:
import static org.junit.Assert.*;
import org.junit.*;

public class LikeTest {
    @Test
    public void simple() {
        Criteria like = new Like(new StringColumn("name", 1), "Joe%");
        assertEquals("name like 'Joe'", like.sqlString());
    }
}

// Like.java
public class Like implements Criteria {
    private StringColumn column;
    private String value;

    public Like(StringColumn column, String value) {
        this.column = column;
        this.value = value;
    }
}

```

```

public String sqlString() {
    return String.format("%s like %s",
        column.getName(), column.sqlValue(value));
}
}

```

Now that both Like and Equals implement the Criteria interface, I can drive the changes into the And class.

```

// AndTest.java
import static org.junit.Assert.*;
import org.junit.*;

public class AndTest {
    @Test
    public void and() {
        Criteria name = new Like(new StringColumn("name", 1), "Joe%");
        Criteria amount = new Equals(new NumericColumn("amount", 5), 5);
        And and = new And(name, amount);
        assertEquals("name like 'Joe%' and amount = 5", and.sql());
    }
}

// And.java
public class And {
    private Criteria left;
    private Criteria right;

    public And(Criteria name, Criteria amount) {
        this.left = name;
        this.right = amount;
    }

    public String sql() {
        return left.sql() + " and " + right.sql();
    }
}

```

So far, you haven't done anything that resembles a composite. Sometimes, the revelation is obvious; other times, it requires a bit more thinking outside the box. Here, if I think about SQL statements for just a few seconds, I realize that And is just another criteria that can be combined with other criteria. For example, I might want a where clause such as:

```
where (amount = 10 and name like 'Joe%') and department = 'Labor'
```

Here is the modified And class:

```

// AndTest.java:
import static org.junit.Assert.*;
import org.junit.*;

public class AndTest {
    @Test
    public void and() {
        Criteria name = new Like(new StringColumn("name", 1), "Joe%");
        Criteria amount = new Equals(new NumericColumn("amount", 5), 5);
        Criteria and = new And(name, amount);
        assertEquals("name like 'Joe%' and amount = 5", and.sqlString());
    }
}

// And.java:
public class And implements Criteria {
    private Criteria left;
    private Criteria right;

    public And(Criteria name, Criteria amount) {
        this.left = name;
        this.right = amount;
    }

    public String sqlString() {
        return left.sqlString() + " and " + right.sqlString();
    }
}

```

Now, I can combine like, equals, and and clauses at will.

The UML diagram shows how the implementation demonstrates use of the Composite design pattern. The key relationship is that the And class is a Criteria implementation, whereas at the same time objects of type And are composed of Criteria objects. Once client code has constructed an appropriate hierarchy of Criteria objects, it can ask the topmost object in the hierarchy for its sqlString in a single call.

Without using the composite design pattern, I can still code a solution, but it's one that would require either more method overloading or if statements, adding to the complexity of the system.

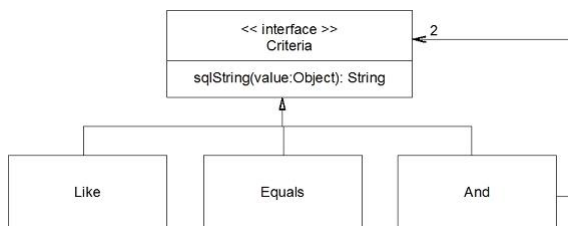



Figure 1: The composite pattern.

Reference

[Gamma] Gamma, E., et. al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

About the Author

Jeff Langr is a veteran software developer celebrating his 25th year of professional software development. He's authored two books and dozens of published articles on software development, including *Agile Java: Crafting Code With Test-Driven Development* (Prentice Hall) in 2005. You can find out more about Jeff at his site, <http://langrsoft.com>, or you can contact him via email at jeff@langrsoft.com.



Check out the Windows Mobile Development Center

Learn how to build great mobile apps for Windows Phone, Windows PCs, & Windows tablets.

GO →

[Sitemap](#) | [Contact Us](#)



Property of Quinstreet Enterprise.
[Terms of Service](#) | [Licensing & Reprints](#) | [About Us](#) | [Privacy Policy](#) | [Advertise](#)
 Copyright 2014 QuinStreet Inc. All Rights Reserved.