Classes

Class Organization:

A class should begin with a list of variables. Public static constants, if any, should come first.

Then private static variables, followed by private instance variables.

Public functions should follow the list of variables. We like to put the private utilities called by a public function right after the public function itself.

Sometimes we need to make a variable or utility function protected so that it can be accessed by a test. For us, tests rule. If a test in the same package needs to call a function or access a variable, we'll make it protected or package scope. However, we'll first look fora way to maintain privacy. Loosening encapsulation is always a last resort.

The name of a class should describe what responsibilities it fulfills. In fact, naming is probably the first way of helping determine class size. If we cannot derive a concise name for a class, then it's likely too large.

The Single Responsibility Principle (SRP) states that a class or module should have one, and only one, reason to change.

We want our systems to be composed of many small classes, not a few large ones. Each small class encapsulates a single responsibility, has a single reason to change, and collaborates with a few others to achieve the desired system behaviors.

We would like cohesion to be high. When cohesion is high, it means that the methods and variables of the class are co-dependent and hang together as a logical whole.

The strategy of keeping functions small and keeping parameter lists short can sometimes lead to a proliferation of instance variables that are used by a subset of methods. When this happens, it almost always means that there is at least one other class trying to get out of the larger class. You should try to separate the variables and methods into two or more classes such that the new classes are more cohesive.

Let's say you want to extract one small part of that function into a separate function. However, the code you want to extract uses four of the variables declared in the function. Must you pass all four of those variables into the new function as arguments?

Not at all! If we promoted those four variables to instance variables of the class, then we could extract the code without passing any variables at all. It would be easy to break the function up into small pieces.

Unfortunately, this also means that our classes lose cohesion because they accumulate more and more instance variables that exist solely to allow a few functions to share them.

When classes lose cohesion, split them! So breaking a large function into many smaller functions often gives us the opportunity to split several smaller classes out as well.

Open-Closed Principle:

Open-Closed Principle: Classes should be open for extension but closed for modification.

Listing 10-9

A class that must be opened for change

```
public class Sql {
   public Sql(String table, Column[] columns)
   public String create()
   public String insert(Object[] fields)
   public String selectAll()
   public String findByKey(String keyColumn, String keyValue)
   public String select(Column column, String pattern)
   public String select(Criteria criteria)
   public String preparedInsert()
   private String columnList(Column[] columns)
   private String valuesList(Object[] fields, final Column[] columns)
   private String selectWithCriteria(String criteria)
   private String placeholderList(Column[] columns)
}
```

The Sqlclass in Listing 10-9 is used to generate properly formed SQL strings given appropriate metadata. It's a work in progress and, as such, doesn't yet support SQL functionality like update statements. When the time comes for the Sqlclass to support an update statement, we'll have to "open up" this class to make modifications. The problem with opening a class is that it introduces risk. Any modifications to the class have the potential of breaking other code in the class. It must be fully retested.

Solution:

Classes

Listing 10-10

A set of closed classes

```
abstract public class Sql {
   public Sql(String table, Column[] columns)
   abstract public String generate();
}

public class CreateSql extends Sql {
   public CreateSql(String table, Column[] columns)
   @Override public String generate()
}

public class SelectSql extends Sql {
   public SelectSql(String table, Column[] columns)
   @Override public String generate()
}

public class InsertSql extends Sql {
   public InsertSql(String table, Column[] columns, Object[] fields)
   @Override public String generate()
   private String valuesList(Object[] fields, final Column[] columns)
}

public class SelectWithCriteriaSql extends Sql {
   public SelectWithCriteriaSql(
      String table, Column[] columns, Criteria criteria)
```

The code in each class becomes excruciatingly simple. Our required comprehension time to understand any class decreases to almost nothing.

Isolating from Change

- A client class depending on concrete details is at risk when those details change
- Using a good base / abstract class can make testing easier as you can create repeatable tests
- Making code more testable makes it more reusable in effect
- DIP, Dependency Inversion Principle The D in SOLID
  - Classes should depend on abstractions, not concrete details