



## CHAPTER 16

# Pair Programming

# 16

### How two people program together

Suppose that Andy, one of our developers, is starting to work on improving the quality of an existing system developed by another team. He joined that team some time ago, so he knows the system fairly well. However, he seems to get stuck. Kent, one of our more senior developers, notices that Andy is doing nothing at his computer and the puzzled look on his face.

Kent: "Hi Andy, what are you doing?"

Andy: "I am trying to get rid of the DataAccesser. But I am not sure how to go about it."

Kent is not very familiar with this system. He asks: "What is DataAccesser? Let me take a look." At the mean time he grabs a chair and sits side by side with Andy.

Side note: Andy and Kent are now programming together using one computer. This is called "pair programming".

Andy opens the DataAccesser class in the IDE. Kent is now faced with lots of code. Kent: "What does it do?"

Andy: "It is used to access the database."

This explanation is too vague. Kent has to look at the fields and methods. Kent: "Let's take a look at its fields."

#### Side note

*When explaining some existing design or code, it is best to show the code first and explain on the way. This is far better than explaining just in words.*

Andy points to the courseData field on the screen with his finger:

```
public abstract class DataAccesser {
    private CourseData courseData;
    private Table table;
    ...
}
```

and says: "CourseData is a weird thing. It contains all the data in the system. I really HATE it! OK, let take a look at it." So he opens the CourseData class in the IDE:

```
public class CourseData {
    ...
    private Students students;
    private Courses courses;
    private Enrollments enrollments;
    private IdCounter idCounter;
    ...
}
```

Then he points to the students field:

```
public class CourseData {
    ...
    private Students students; 🐭 🐭 🐭
    private Courses courses;
    private Enrollments enrollments;
    private IdCounter idCounter;
    ...
}
```

and explains: "For example, students inherits from DataAcesser and represents all the students in the database." At the same time he opens the Students class in the IDE:

```
public class Students extends DataAcesser {
    ...
}
```

Returning to the CourseData class and pointing to the courses field:

```
public class CourseData {
    ...
    private Students students;
    private Courses courses; 🐭 🐭 🐭
    private Enrollments enrollments;
    private IdCounter idCounter;
    ...
}
```

Andy continues: "Similarly, it represents all the courses in the database; The enrollments field represents all the enrollments in the database and etc."

Kent: "OK, I see. Let's return to DataAcesser."

Andy shows DataAcesser again. Kent points to the table field in DataAcesser:

```
public abstract class DataAcesser {
    private CourseData courseData;
    private Table table; 🐭 🐭 🐭
    ...
}
```

and asks: "What's this?"

Andy: "The Table class is simple. It contains the table name and the names of its fields."

Kent: "OK." Now Kent understands that a DataAccesser has a Table and can reference all the other DataAccessers in the system. Kent continues to ask: "What methods it has?"

Andy scrolls down from start to end to find an interesting method and finds one. He points to the deleteAll method:

```
public void deleteAll() throws DataAccessException {
    Vector refs = getRefsAccessersForDeleteAll();
    if (refs != null)
        for (int i = 0; i < refs.size(); i++)
            ((DataAccesser) refs.elementAt(i)).deleteAll();
    PreparedStatement st;
    try {
        st =
            getConnection().prepareStatement("DELETE FROM " + table.getName());
        try {
            executeUpdate(st);
        } finally {
            st.close();
        }
    } catch (SQLException e) {
        throw new DataAccessException(e);
    }
}
```

and says: "For example, the deleteAll method deletes all records in the table."

The call to the getRefsAccessersForDeleteAll method is drawing Kent's attention. The rest of the method is pretty familiar to him. He points to the line:

```
public void deleteAll() throws DataAccessException {
    Vector refs = getRefsAccessersForDeleteAll();
    if (refs != null)
        for (int i = 0; i < refs.size(); i++)
            ((DataAccesser) refs.elementAt(i)).deleteAll();
    ...
}
```

and asks: "What's this?"

Andy: "It gets all the other DataAccessers that refer to this DataAccesser." Andy points to the loop:

```
public void deleteAll() throws DataAccessException {
    Vector refs = getRefsAccessersForDeleteAll();
    if (refs != null)
        for (int i = 0; i < refs.size(); i++)
            ((DataAccesser) refs.elementAt(i)).deleteAll();
    ...
}
```

```
}

```

and explains: "Here it will call deleteAll on those DataAccessers first."

Kent doesn't really understand what Andy said. So he asks: "Would you give an example?"

### **Side note**

*When we don't understand what someone is saying, the best thing to do is to ask him for an example. This is probably the most important skill in communication (and pair programming).*

Andy: "Sure. For example, if you are going to delete all students, because there may be enrollments referring to the students, it will delete all the enrollments first."

Now Kent comes to conclude that a DataAccesser represents a table in a database. The table is "smart" in that it can perform cascade deletes. "I see. Let's look at another method." says Kent.

Andy: "OK." Then he continues to scroll down to find another interesting method. "Here you are." He points to a method named "update":

```
public int update(Object[][] fieldsAndValues, Object[][] keyFieldsAndValues)
    throws DataAccessException {
    try {
        PreparedStatement st =
            getConnection().prepareStatement(
                "UPDATE "
                + table.getName()
                + " SET "
                + getParameterString(fieldsAndValues)
                + getConditionString(keyFieldsAndValues, "="));
        try {
            try {
                setParameters(st, fieldsAndValues, 1);
                setParameters(
                    st,
                    keyFieldsAndValues,
                    fieldsAndValues.length + 1);
            } catch (InvalidArgumentException e1) {
                e1.printStackTrace();
            }
            return executeUpdate(st);
        } finally {
            st.close();
        }
    } catch (SQLException e) {
        throw new DataAccessException(e);
    }
}
```

Andy explains: "This method updates some records in the table." He goes on to point to the fieldsAndValues and keyFieldsAndValues parameters:

```

public int update(Object[][] fieldsAndValues, Object[][] keyFieldsAndValues)
    throws DataAccessException {
    ...
}

```

and says: "They are something that I hate very much. Let me show you how they are used." Then he uses the IDE to search for a call to the update method. "Got it. Take a look at this.", he says.

```

private void updateStudentAnyway(Student student) throws DataAccessException {
    Object[][] fieldsAndValues = { {
        StudentsTable.eFirstNameField, student.getEFirstName(), {
        StudentsTable.cFirstNameField, student.getCFirstName()
    }, {
        StudentsTable.eLastNameField, student.getELastName()
    }, {
        StudentsTable.cLastNameField, student.getCLastName()
    }, {
        StudentsTable.idTypeField, student.getIdType()
    }, {
        StudentsTable.idNoField, student.getIdNo()
    }, {
        StudentsTable.nationalityField, student.getNationality()
    }, {
        StudentsTable.genderField, new Boolean(student.isMale())
    }, {
        StudentsTable.birthDateField, student.getBirthDate()
    }, {
        StudentsTable.regionField, student.getRegionId()
    }, {
        StudentsTable.eAddressField, student.getEAddress()
    }, {
        StudentsTable.cAddressField, student.getCAddress()
    }, {
        StudentsTable.telField, student.getTel()
    }, {
        StudentsTable.mobileField, student.getMobile()
    }, {
        StudentsTable.faxField, student.getFax()
    }, {
        StudentsTable.emailField, student.getEmail()
    }
    };
    Object[][] keyFieldsAndValues = { { StudentsTable.studentNoField,
    student.getStudentNo() }
    };
    update(fieldsAndValues, keyFieldsAndValues);
}

```

Andy: "Look, this method is calling update to update a student." Then he points to:

```

private void updateStudentAnyway(Student student) throws DataAccessException {
    Object[][] fieldsAndValues = { {
        StudentsTable.eFirstNameField, student.getEFirstName(), {
        StudentsTable.cFirstNameField, student.getCFirstName()
    },
    ...
}

```

and says: "It is a 2D array holding the fields to be set and their new values." Then he points to:

```
private void updateStudentAnyway(Student student) throws DataAccessException {
    Object[][] fieldsAndValues = { {
        ...
    } };
    Object[][] keyFieldsAndValues = { { StudentsTable.studentNoField,
    student.getStudentNo() } };
    update(fieldsAndValues, keyFieldsAndValues);
}
```

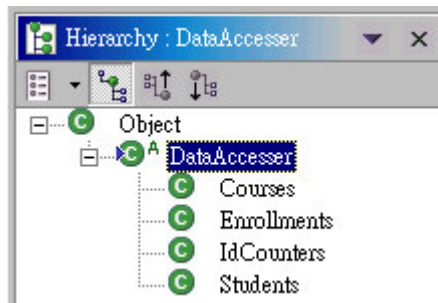
and says: "It is also a 2D array. It specifies the SQL condition. The 2D arrays suck."

Kent agrees that the 2D arrays suck and is thinking to replace each with a list of field-value pairs. "Right, we'll get rid of them." He thinks that he has seen enough of the code. "So, you'd like to get rid of DataAccessor?"

Andy: "Yeah, I think it is ugly." He is waiting for Kent's advice.

After thinking for a while about to get rid of DataAccessor, Kent says: "OK. Let's take a look at the simplest class that is extending DataAccessor."

Andy: "OK." He right clicks DataAccessor and choose "Open Type Hierarchy" in the IDE and instantly all the derived classes of DataAccessor are listed:



Kent is impressed by this powerful feature of the IDE. He keeps it in mind for future use.

#### **Side note**

*Knowledge has just been transferred. As shown in this case, a more experienced developer can still learn from a less experienced one.*

Andy points to a class named "IdCounters" and says "This should be the simplest one." Then he opens it in the IDE.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from [www.agileskills.org](http://www.agileskills.org)

Kent browses the code of IdCounters and finds that it only uses a few methods of DataAccesser. "OK. Let's do it. Let's create a DBTable class."

Andy doesn't understand what Kent is trying to do. "OK, but what are you trying to do?"

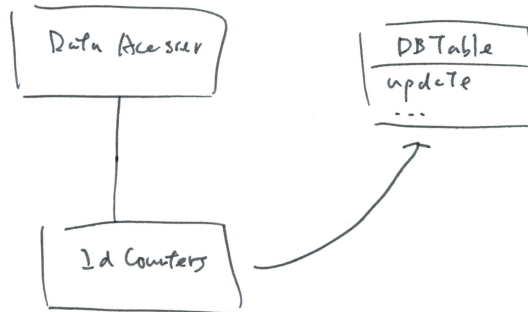
Kent: "At the moment IdCounters is inheriting DataAccesser. Right?" Kent draws a sketch on a piece of scrap paper:



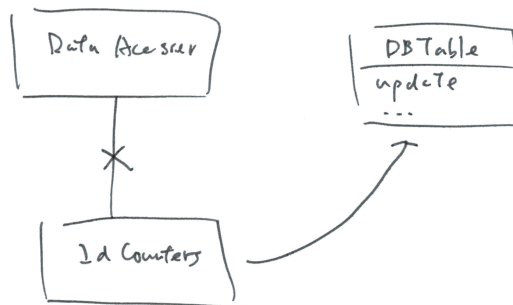
#### Side note

*Kent is trying to show Andy how the design in his mind works. Usually, the best way to communicate a design is to show him the code and explain on the way. However, as there is no code to show yet, diagrams can be a good tool in explanations. Note that the diagram above looks like UML but doesn't really obey the UML rules (e.g., in UML inheritance is represented by a line ended with a hollow arrow). This is OK because the inheritance relationship is already clearly spoken in the conversation.*

He continues: "I'd like to create a DBTable class to provide the functionality of DataAccesser and let IdCounters use a DBTable instead of inheriting DataAccesser." He continues to draw:



"In the process I don't want to change DataAccesser at all. When IdCounters no longer uses any of DataAccesser's service, we can break the inheritance." He crosses out the inheritance line in the drawing:



#### Side note

*Again, UML rules are not obeyed. There is no such a cross symbol in UML to mean canceling an inheritance. But it is so obvious to an average person.*

Andy now feels much more happier because he knows that he can replace the reliance on DataAccesser bit by bit. He is excited: "Great! Let's do it."

**Side note**

- *Andy and Kent have just performed some design together. In pair programming, designing together is one of the major activities. They have also worked out a refactoring strategy together. Andy offered his knowledge on the classes in the system and suggested that DataAccesser was ugly and should be replaced (what to refactor). Based on Andy's knowledge Kent suggested the strategy to reduce the reliance on DataAccesser bit by bit by migrating the clients to use DBTable gradually (how to refactor).*
- *In pair programming, different people with different knowledge/skills can combine their skills to solve a single difficult problem. For example, Andy knows the system but not as good in OO and refactoring, while Kent is better in OO and refactoring but knows little about the system. Obviously, letting either of them refactor the system alone will be extremely difficult. But if they work together, their knowledge is combined and refactoring the system becomes much easier. The above scenario is just one kind of fruitful combination. There are others such as: Database administrator plus programmer (optimizing code to access the database), UI designer plus programmer (developing UI), fellow programmer plus fellow programmer (one is developing code that uses the code developed by the other), junior programmer plus designer/architect (implementing the design or checking if the design works) and etc.*
- *In pair programming, knowledge and best practices are frequently transferred. Now Andy knows more about how to refactor in small steps and Kent knows more about the existing system.*
- *In pair programming, programmers are more confident and happier. For example, Andy was very worried at the beginning. After pairing with Kent, he is much happier. In fact, Kent also feels more confident after having Andy double check his ideas and show the way around the system.*

Kent: "Good. Let's create the DBTable class."

Andy looks puzzled. He asks: "Shouldn't we start with a failing test first?"

Kent: "Ah, you're right! Let's do the test first."

So Andy creates a DBTableTest class:

```
public class DBTableTest extends TestCase {  
}
```

**Side note**

*Andy and Kent are going to write code to test DBTable. Right, in addition to designing together, testing together is another major activity in pair programming.*

Andy starts typing a method to test the deleteAll method. He types "public", and then a method name starting with "test":

```
public class DBTableTest extends TestCase {
    public test
}
```

Kent points to the location:

```
public class DBTableTest extends TestCase {
    public test
}
```

and remind him: "void is missing."

#### **Side note**

*Kent detected and corrected the bug immediately after it was created. Right, in addition to designing together and testing together, debugging together is another major activity in pair programming.*

Andy promptly corrects the error and then continues:

```
public class DBTableTest extends TestCase {
    public void test
}
```

Andy: "test what? How about testDeleteAll?"

Kent: "Fine." Andy types it in:

```
public class DBTableTest extends TestCase {
    public void testDeleteAll() {
    }
}
```

Andy: "Then we should create a DBTable."

```
public class DBTableTest extends TestCase {
    public void testDeleteAll() {
        DBTable table = new DBTable();
    }
}
```

Andy: "Then we can call deleteAll."

```
public class DBTableTest extends TestCase {
    public void testDeleteAll() {
        DBTable table = new DBTable();
        table.deleteAll();
    }
}
```

Kent has made no comments for a while. He has been thinking about how to test deleteAll and has come to a solution.

Andy: "So, how to test it?"

Kent: "Let's create an interface to simulate the database server. It will can execute an SQL statement sent to it."

Andy has never seen unit testing done this way. Therefore he doesn't really understand what Kent is saying. He says: "Sorry, I don't understand."

Kent: "To test DBTable, it seems that we need to setup a database and a test table. This is troublesome. In addition, running the tests will take a long time. Therefore, we should instead simulate the database."

Andy still doesn't really understand because the description is too abstract. He says: "I am really sorry, but I still don't understand."

Kent says: "It's OK. Just type what I say. You will understand very soon. Now, create an interface named DBServer." Andy creates it as told:

```
public interface DBServer {  
}
```

and is ready to type the first method signature.

#### Side note

*If we just can't communicate a proposed design to another person, it is best to just type the code. Who should type the code? Let the weaker partner do the typing. If Kent had done the typing, Andy would have complained about not knowing what had been going on or simply have fallen asleep.*

Kent says: "public, int, executeUpdate, parenthesis." Andy types accordingly:

```
public interface DBServer {  
    public int executeUpdate()  
}
```

Note that the IDE automatically adds the closing parenthesis. Kent continues, "string, SQL. That's it." Andy types accordingly:

```
public interface DBServer {  
    public int executeUpdate(String sql);  
}
```

Kent says: "Return to the test." Andy does it accordingly:

```
public class DBTableTest extends TestCase {
    public void testDeleteAll() {
        DBTable table = new DBTable();
        table.deleteAll();
    }
}
```

Kent points to the line:

```
public class DBTableTest extends TestCase {
    public void testDeleteAll() {
        DBTable table = new DBTable();
        ☞ ☞ table.deleteAll();
    }
}
```

and says: "Add a line before it." Andy does it accordingly. Kent continues, "table, dot, setDBServer, parenthesis." Andy types accordingly:

```
public class DBTableTest extends TestCase {
    public void testDeleteAll() {
        DBTable table = new DBTable();
        table.setDBServer()
        table.deleteAll();
    }
}
```

Kent continues, "new, DBS, auto-complete, parenthesis." Kent is using the auto-complete feature of the IDE to expand "DBS" to "DBServer". Andy uses the auto-complete feature day in and day out, so he has no trouble using it. Actually "DBS" matches a few other classes the system like DBSanityChecker, DBSuperUser, but Andy quickly chooses DBServer without being told at all:

```
public class DBTableTest extends TestCase {
    public void testDeleteAll() {
        DBTable table = new DBTable();
        table.setDBServer(new DBServer())
        table.deleteAll();
    }
}
```

It means that Andy is actively engaged and is following closely what Kent is trying to do.

Kent points to the following location:

```
public class DBTableTest extends TestCase {
    public void testDeleteAll() {
        DBTable table = new DBTable();
        table.setDBServer(new DBServer(☞))
        table.deleteAll();
    }
}
```

and says: "Go to here. Auto-complete." Kent is using the auto-complete feature to generate a

skeleton implementation of an interface. Andy didn't know that it can be used this way, but he does it anyway:

```
public class DBTableTest extends TestCase {
    public void testDeleteAll() {
        DBTable table = new DBTable();
        table.setDBServer(new DBServer() {
            public int executeUpdate(String sql) {
                return 0;
            }
        })
        table.deleteAll();
    }
}
```

"Wow! This is great! Only if I knew it earlier!", Andy is impressed.

#### Side note

*Knowledge is transferred again. After learning this trick, Andy's productivity is increased for the rest of his life. If he later pairs with other colleagues, he will also transfer this trick to them. This way, knowledge is spread throughout the whole organization.*

Kent points to the line:

```
public class DBTableTest extends TestCase {
    public void testDeleteAll() {
        DBTable table = new DBTable();
        table.setDBServer(new DBServer() {
            public int executeUpdate(String sql) {
                return 0;
            }
        })
        table.deleteAll();
    }
}
```

and says: "Add a line before it." Andy does it accordingly:

```
public class DBTableTest extends TestCase {
    public void testDeleteAll() {
        DBTable table = new DBTable();
        table.setDBServer(new DBServer() {
            public int executeUpdate(String sql) {
                return 0;
            }
        })
        table.deleteAll();
    }
}
```

Kent continues, "assertEquals, sql, double quote, delete, from."

```

public class DBTableTest extends TestCase {
    public void testDeleteAll() {
        DBTable table = new DBTable();
        table.setDBServer(new DBServer() {
            public int executeUpdate(String sql) {
                assertEquals(sql, "delete from")
                return 0;
            }
        })
        table.deleteAll();
    }
}

```

Kent hesitates for a second and then points to the location:

```

public class DBTableTest extends TestCase {
    public void testDeleteAll() {
        DBTable table = new DBTable();
        table.setDBServer(new DBServer() {
            public int executeUpdate(String sql) {
                assertEquals(sql, "delete from")
                return 0;
            }
        })
        table.deleteAll();
    }
}

```

and says: "Pass a string 'abc' to it." Andy does it accordingly:

```

public class DBTableTest extends TestCase {
    public void testDeleteAll() {
        DBTable table = new DBTable("abc");
        table.setDBServer(new DBServer() {
            public int executeUpdate(String sql) {
                assertEquals(sql, "delete from")
                return 0;
            }
        })
        table.deleteAll();
    }
}

```

Kent points to the location:

```

public class DBTableTest extends TestCase {
    public void testDeleteAll() {
        DBTable table = new DBTable("abc");
        table.setDBServer(new DBServer() {
            public int executeUpdate(String sql) {
                assertEquals(sql, "delete from")
                return 0;
            }
        })
        table.deleteAll();
    }
}

```

and says: "from, space, abc." Andy types accordingly and now understands that "abc" is the table name:

```
public class DBTableTest extends TestCase {
    public void testDeleteAll() {
        DBTable table = new DBTable("abc");
        table.setDBServer(new DBServer() {
            public int executeUpdate(String sql) {
                assertEquals(sql, "delete from abc");
                return 0;
            }
        })
        table.deleteAll();
    }
}
```

Andy asks: "Why not call it 't'? I think it is better than 'abc'." Kent think "t" is indeed more descriptive than "abc". So, he says: "Yeah, that's a good idea." So, Andy changes it to "t":

```
public class DBTableTest extends TestCase {
    public void testDeleteAll() {
        DBTable table = new DBTable("t");
        table.setDBServer(new DBServer() {
            public int executeUpdate(String sql) {
                assertEquals(sql, "delete from t");
                return 0;
            }
        })
        table.deleteAll();
    }
}
```

#### Side note

- *After typing all the code dictated by Kent, Andy finally understands how to use DBServer to simulate a database to unit test DBTable. This is a technique called "mock objects". Knowledge is transferred once again.*
- *Andy and Kent just made another design decision together regarding the table name. But this time it was Andy who made the suggestion. It means that even though one person may be very good at something (e.g., design), there is still room for others to make significant contributions.*

However, there is a syntax error detected by the IDE:

```

public class DBTableTest extends TestCase {
    public void testDeleteAll() {
        DBTable table = new DBTable("t");
        table.setDBServer(new DBServer() {
            public int executeUpdate(String sql) {
                assertEquals(sql, "delete from t");
                return 0;
            }
        });
        table.deleteAll();
    }
}

```

While Kent is checking what's wrong, Andy already finds the problem: "Oops, a semicolon is missing!" So he corrects it:

```

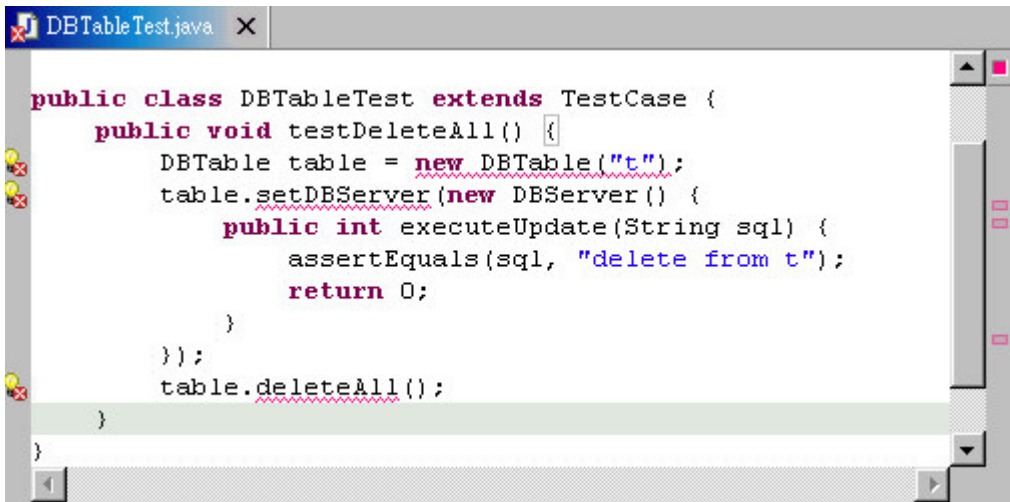
public class DBTableTest extends TestCase {
    public void testDeleteAll() {
        DBTable table = new DBTable("t");
        table.setDBServer(new DBServer() {
            public int executeUpdate(String sql) {
                assertEquals(sql, "delete from t");
                return 0;
            }
        });
        table.deleteAll();
    }
}

```

#### **Side note**

This time Andy found the bug first. It means that being more experienced doesn't necessarily mean that he can always find the bug faster. Each person is good at detecting certain types of bugs but poor at others. Fortunately, the "blind zone" of one person usually doesn't overlap much with that of another. For example, Andy didn't notice that he missed "void" when creating the testDeleteAll method, but this was very obvious to Kent. In contrast, Kent didn't notice that a semicolon was missing, but this was quite visible to Andy. Without a pair partner, it would have taken much longer for each of them to find the problems.

Then the IDE detects other syntax errors:



```
public class DBTableTest extends TestCase {
    public void testDeleteAll() {
        DBTable table = new DBTable("t");
        table.setDBServer(new DBServer() {
            public int executeUpdate(String sql) {
                assertEquals(sql, "delete from t");
                return 0;
            }
        });
        table.deleteAll();
    }
}
```

To get rid of these errors, Andy and Kent define the DBTable constructor, the setDBServer method and an empty deleteAll method:

```
public class DBTable {
    private String tableName;
    private DBServer dbServer;

    public DBTable(String tableName) {
        this.tableName = tableName;
    }
    public void setDBServer(DBServer dbServer) {
        this.dbServer = dbServer;
    }
    public void deleteAll() {
    }
}
```

They run the test and it passes!

#### Side note

*As something unusual has occurred, they will debug together again.*

Kent has made this kind of mistakes before, so he finds the problem in just a few seconds. He points to the line:

```
public void testDeleteAll() {
    DBTable table = new DBTable("t");
    table.setDBServer(new DBServer() {
        public int executeUpdate(String sql) {
            assertEquals(sql, "delete from t");
            return 0;
        }
    });
}
```

```

    });
    table.deleteAll();
}

```

and says: "Add a line before it." Andy does it accordingly:

```

public void testDeleteAll() {

    DBTable table = new DBTable("t");
    table.setDBServer(new DBServer() {
        public int executeUpdate(String sql) {
            assertEquals(sql, "delete from t");
            return 0;
        }
    });
    table.deleteAll();
}

```

Then Kent says: "final, StringBuffer, callLog, equal, new, StringBuffer" and Andy types accordingly:

```

public void testDeleteAll() {
    final StringBuffer callLog = new StringBuffer();
    DBTable table = new DBTable("t");
    table.setDBServer(new DBServer() {
        public int executeUpdate(String sql) {
            assertEquals(sql, "delete from t");
            return 0;
        }
    });
    table.deleteAll();
}

```

Then Kent points to the line:

```

public void testDeleteAll() {
    final StringBuffer callLog = new StringBuffer();
    DBTable table = new DBTable("t");
    table.setDBServer(new DBServer() {
        public int executeUpdate(String sql) {
            assertEquals(sql, "delete from t");
            return 0;
        }
    });
    table.deleteAll();
}

```

and says: "Add a line before it." Andy does it accordingly:

```

public void testDeleteAll() {
    final StringBuffer callLog = new StringBuffer();
    DBTable table = new DBTable("t");
    table.setDBServer(new DBServer() {
        public int executeUpdate(String sql) {

            assertEquals(sql, "delete from t");

```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from [www.agileskills.org](http://www.agileskills.org)

```

        return 0;
    }
});
table.deleteAll();
}

```

Kent says: "callLog, dot, append, double quote, x, double quote" and Andy types accordingly:

```

public void testDeleteAll() {
    final StringBuffer callLog = new StringBuffer();
    DBTable table = new DBTable("t");
    table.setDBServer(new DBServer() {
        public int executeUpdate(String sql) {
            callLog.append("x");
            assertEquals(sql, "delete from t");
            return 0;
        }
    });
    table.deleteAll();
}

```

Kent points to the following line:

```

public void testDeleteAll() {
    final StringBuffer callLog = new StringBuffer();
    DBTable table = new DBTable("t");
    table.setDBServer(new DBServer() {
        public int executeUpdate(String sql) {
            callLog.append("x");
            assertEquals(sql, "delete from t");
            return 0;
        }
    });
    table.deleteAll();
}

```

and is about to tell Andy to add a line after it, but Andy already knows what he is trying to do and add the line by himself:

```

public void testDeleteAll() {
    final StringBuffer callLog = new StringBuffer();
    DBTable table = new DBTable("t");
    table.setDBServer(new DBServer() {
        public int executeUpdate(String sql) {
            callLog.append("x");
            assertEquals(sql, "delete from t");
            return 0;
        }
    });
    table.deleteAll();
    assertEquals(callLog.toString(), "x");
}

```

#### Side note

*After typing the code, Andy now understands how to use a call log to track the call sequence. Knowledge is transferred once again.*

They run the test and now it fails. They feel that they have made good progress.

### Side note

*Now they are about to implement deleteAll in DBTable. Right, in addition to designing together and testing together, coding together is another major activity in pair programming.*

Andy says: "OK, let's copy the code from DataAccesser."

```
public void deleteAll() throws DataAccessException {
    Vector refs = getRefsAccessersForDeleteAll();
    if (refs != null)
        for (int i = 0; i < refs.size(); i++)
            ((DataAccesser) refs.elementAt(i)).deleteAll();
    PreparedStatement st;
    try {
        st =
            getConnection().prepareStatement("DELETE FROM " + table.getName());
        try {
            executeUpdate(st);
        } finally {
            st.close();
        }
    } catch (SQLException e) {
        throw new DataAccessException(e);
    }
}
```

Andy deletes the code about cascade deletes and then uses DBServer instead of the JDBC connections and prepared statements:

```
public void deleteAll() {
    dbServer.executeUpdate("DELETE FROM " + tableName);
}
```

Then they run the test again, but it still fails! Kent points to the line:

```
public void testDeleteAll() {
    final StringBuffer callLog = new StringBuffer();
    DBTable table = new DBTable("t");
    table.setDBServer(new DBServer() {
        public int executeUpdate(String sql) {
            callLog.append("x");
            assertEquals(sql, "delete from t");
            return 0;
        }
    });
    table.deleteAll();
    assertEquals(callLog.toString(), "x");
}
```

and says: "Let's set a breakpoint here." However, Andy suddenly shouts, "Ah! The character cases are different! The test is using lowercase but the code is using uppercase!" So they change the test to:

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from [www.agileskills.org](http://www.agileskills.org)

```
public void testDeleteAll() {
    final StringBuffer callLog = new StringBuffer();
    DBTable table = new DBTable("t");
    table.setDBServer(new DBServer() {
        public int executeUpdate(String sql) {
            callLog.append("x");
            assertEquals(sql.toLowerCase(), "delete from t");
            return 0;
        }
    });
    table.deleteAll();
    assertEquals(callLog.toString(), "x");
}
```

Then they run the test and it passes. Now the task is done and they run all the existing tests. During this time, they take a break to go get a drink (also done together because they are now a better team).

#### Side note

- Because pair programming is a mentally intensive exercise, regular breaks are good for the productivity.
- After performing this pair programming session, both Andy and Kent are familiar with this part of the system (DBTable, DBServer, etc.). If anyone of them is on vacation or leaves the company, the other can still maintain it. It means pair programming enhances the company's resistance to staff turn-over.

Later, another colleague John comes over and asks: "Andy, I'll start to work on the transcript printing story. I know that you wrote the part about the inputting of marks, would you like to pair with me?"

Andy: "Sure." At the same time Andy moves to pair with John. Kent is now free to pair with someone else.

#### Side note

- *It is good to switch partners regularly (probably at the start of a new task). Now Andy will further transfer the knowledge he learned to John. He will also learn something from John (e.g., how transcript printing works).*
- *As Andy knows how the marks are saved, he can probably help in retrieving the marks to print transcripts. Therefore, it is wise for John to ask Andy to pair with him to work on this task.*

## Summary on the benefits of pair programming

- Combine different knowledge to tackle a difficult task.
- Knowledge transfer.
- Efficient bug detection and removal.
- Programmers are happier.
- Resistance to staff turn-over.

## Summary on the skills for pair programming

- Use code to explain existing designs.
- Use examples to explain.
- Use diagrams to explain proposed designs.
- If you just can't communicate a proposed design, type the code in.
- Let the weaker partner do the typing to keep him engaged.
- Take a break regularly.
- Switch partners regularly.

## Do we have to double the number of programmers

Even though pair programming has quite many benefits, if we let two programmers work on a single task, do we have to double the number of programmers?

A research conducted by Laurie Williams shows that in a particular university environment but otherwise without any particular arrangements, it takes 15% more time for a pair to do as much work as two individuals working alone. It means we don't have to double the number of programmers, but just 15% more programmers or let the same number of programmers work for 15% longer.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from [www.agileskills.org](http://www.agileskills.org)

So, on one hand pair programming has some great benefits, but at the same time it needs 15% more development time. Is it worth it? The above research shows that the software delivered by a pair contains 15% fewer bugs than that of the individuals. These bugs will have to be fixed. Based on industry statistics, Alistair Cockburn and Laurie Williams conclude that the extra time the individuals will take to fix those bugs is 15-60 times of that 15% extra development time spent by the pair. So, just one benefit of pair programming (more efficient bug removal) alone already justifies the 15% increase in development time.

## When will pair programming not work

Pair programming doesn't always work. Because it requires two people communicating and making decisions together, if somehow communication or decision making is not happening, then pair programming will not work.

When will communication not happen? For example, suppose that Kent is busying working hard to meet the deadline for his project, but the director of development Paul insists that he help Andy. So Kent reluctantly pairs with Andy. He asks Andy to show him the code. After seeing that, although he is not happy with DataAccesser either, he is so concerned with his own deadline that he says: "Oh my! It is a lot of work to remove it. You can simply rename it to DBTable." Andy says: "No! The current system is so database centric that it is too difficult to work on it any more. We must do something!" Paul says that you would help me...' After that, when Andy is typing, Kent is not paying attention at all. He is not giving any advice either.

In this case above, communication is not happening because Kent is reluctant to pair with Andy. They seem to be pairing but in fact they are not. They don't share a common goal (i.e., improving the quality of that system). When people don't share a common goal, they have no incentive to communicate.

This is not the only possible reason for lack of communication. For example, if a partner is emotionally rejecting everything communicated, then there is no communication. When Andy and Kent were working on the testDeleteAll method, Andy suggested to use "t" instead of "abc" as the table name. What if Kent had responded like this: "abc is just fine. I have been using this kind of constants in tests for years. How dare you challenge me? You are just a fresh graduate. What do you know? When I started programming, you were still in kindergarten."

The problem may also occur at the sending end of the communication path. That is, a partner lacks confidence so much that he doesn't dare to raise anything. When Andy and Kent were starting to write DBTable, Andy (rightly) questioned Kent why he suggested writing DBTable without writing a failing test first. What if Andy had been so impressed by Kent's seniority and OO skills that he hadn't questioned him? Then the system would not have benefited from his valid suggestions.

Lack of confidence is just one reason for not raising anything. Another one is to avoid looking stupid. When Andy was explaining to Kent about the cascade delete behavior in the deleteAll method of DataAccesser, Kent didn't understand his explanation. So he asked Andy to give an example. What if Kent had felt that he would look stupid to not understand such a simple method? Then Kent would not have understood the system to suggest a good way to refactor it.

In summary, here are some common problems that can stop pair programming from working:

- Reluctant pairing.
- Rejecting every single suggestion from the partner or attacking or demeaning the partner.
- Not making suggestions to avoid "challenging" the "gurus".
- Not asking questions to avoid looking stupid.

What if these problems really occur? As these have more to do with management and personality than technical skills, the best way is probably to allow them not to pair or to pair with somebody else (e.g., guru plus guru).

## References

- Costs and benefits of pair programming:
  - Laurie Williams' dissertation at <http://www.cs.utah.edu/~lwilliam/Papers/dissertation.pdf> details a research done in a university environment regarding pair programming.
  - Alistair Cockburn and Laurie Williams' paper at <http://collaboration.csc.ncsu.edu/laurie/Papers/XPSardinia.PDF> analyzes the costs and benefits of pair programming.
  - <http://www.cs.utah.edu/~lwilliam/Papers/ieeeSoftware.PDF>.
  - <http://www.agilealliance.org/articles/articles/QuantitativeAssessment.pdf>.
- How to pair program effectively:
  - <http://www.agilealliance.org/articles/articles/Kindergarten.pdf>.
  - <http://www.pairprogramming.com/conversantpairing.htm>.

---

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from [www.agilekills.org](http://www.agilekills.org)

- <http://www.agilealliance.org/articles/articles/PairedProgrammingandPersonalityTraits.pdf>
- Demonstrations:
  - Robert Martin and Robert Koss' article at <http://www.objectmentor.com/publications/xpepisode.htm> demonstrates how Extreme Programming, including pair programming, works in practice.
- Further references:
  - Laurie Williams' web site about pair programming: <http://www.pairprogramming.com/>.
  - A website about Agile Development: <http://www.agilealliance.org>.

## Chapter exercises

### Problems

---

1. Find someone to pair with you to develop an "update" method in DBTable that models after its counterpart in DataAccesser without using arrays.