Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science 157 (2006) 119–132

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Improving the Quality of Web-based Enterprise Applications with Extended Static Checking: A Case Study

Frédéric Rioux[1] , Patrice Chalin[1]

Dependable Software Research Group (DSRG)
Department of Computer Science and Software Engineering
Concordia University
Montréal, Canada

Abstract

ESC/Java2 is a tool that statically detects errors in Java programs and that uses the Java Modeling Language (JML) as its annotation language. ESC/Java2 can modularly reason about the code of a Java Web-based Enterprise Application (WEA) and uncover potential errors. In this paper, we assessed the effectiveness of ESC/Java2 at helping developers increase WEA quality by detecting design and implementation issues.

Keywords: Web-based Enterprise Application, Extended Static Checking, Design by Contract, Java Modeling Language

1 Introduction

The evolution of programming languages has allowed software engineers to develop increasingly larger software systems while maintaining, or improving, product quality. This has been achieved in part by increasing the level of abstraction of the languages, exploiting new paradigms (such as object-orientedness), and enabling compilers to perform static checks and/or embed run-time checking code when the former is neither possible nor practical. In light of the above, one may ask: what will be the next programming language

[1] Email: {f_rioux, chalin}@cse.concordia.ca

doi:10.1016/j.entcs.2005.12.050

and tool advances that are likely to go mainstream? We believe that for most domains of application, these advances are likely to include Design by Contract (DBC) and Extended Static Checking (ESC). DBC can gradually be integrated into projects as a lightweight formal method allowing developers to become accustomed to the method. Among others, lint-like [7] ESC tools — familiar to many programmers — can then make use of this extra formal documentation to perform rapid and effective checks.

Enterprise Applications (EAs) are generally characterized by the large volume of data they manipulate, the business rules they embed, and their need to integrate with other (often legacy) applications [6]. An important class of EAs are web-based enterprise applications (WEAs) that businesses and consumers are increasingly coming to make use of — in the third quarter of 2004 alone, retail e-commerce sales in the US have been estimated at $17.6 billion US, an increase of 4.7% from the previous quarter. This figure has been on the rise since 1999 [16].

This paper presents the preliminary results of a case study whose main goal is to assess the effectiveness of ESC/Java2 [1][2][4] at helping developers increase WEA quality by detecting design and implementation issues. In doing so, we wrote lightweight JML specifications for some *javax* and *java.sql* classes that can be reused not only by ESC/Java2 users, but by the whole JML community. The subject of our study is a collection of WEAs mainly based on a small framework (named *SoenEA*) that has been in use at Concordia for almost two years now in courses where WEA architecture and design is being taught.

This paper is organized as follows. DBC, ESC and ESC/Java2 are covered in Section 2. Section 3 reiterates our goals and presents the case study including an explanation for our choice of application domain. The remaining sections offer a discussion of the case study results, future work and conclusion.

2 Background

2.1 Design by Contract (DBC)

DBC is an approach to design which views the relationship between two classes — a supplier and a client — as a formal agreement, or a contract [12]. Such an agreement expresses each party's rights and obligations. Contracts usually take the form of preconditions, post-conditions, and invariants.

Contracts are a form of module specification. We believe that it is possible to raise our level of trust of large scale and complex systems if their contracts are unambiguous, correct, and verifiable. DBC is currently supported by the Eiffel programming language and some specification languages, including

```
//@ requires y >= 0;
public static int isqrt(int y){
  ...
}
```

Fig. 1. Sample lightweight specification

Behavioral Interface Specification Languages (BISLs).

A BISL is a language that can be used to accurately describe a module's interface, hence, by definition, implementing DBC. The main characteristic of a BISL is that it is tailored to a particular programming language. BISLs are an interesting kind of specification language, since they can be used to bridge the gap between the specification and design activities of the software development lifecycle. They also set the basis for powerful and automatic verification. Moreover, they are flexible, allowing them to be introduced incrementally, as needed, to new developments as well as legacy code.

The Java Modeling Language (JML) is a Java BISL that is actively being developed by an international team of researchers who collaborate on the language definition and tool implementation. Details on the latest tools and applications can be found in [1]. Use of JML for DBC is described in [8].

JML specifications can be embedded in Java source files as specially formatted comments or in external specification files. A JML expression looks like a Java expression since JML keeps most of Java's syntax and semantics [10]. Such a tight coupling between Java and JML lowers the burden required of developers to learn and use JML.

JML supports both lightweight and heavyweight specifications. Lightweight specifications are less detailed and complete than heavyweight specifications and are often composed of individual clauses, describing only one aspect of the intended behavior. JML was designed to support lightweight specifications and has semantics that allow most clauses to be omitted.

In JML, method specifications are usually written just before the methods. Preconditions are represented by a *requires* clause and post-conditions by an *ensures* clause. Every clause is a Boolean expression. Method calls can be used in the specification, but they must be calls to methods declared as *pure*, i.e. that they have no side-effects [9].

Figure 1 and Figure 2 show partial and more detailed specifications, respectively, of an integer square root function. In the first case, the only constraint is that the number must be a positive integer; however, in the later case, the function is also guaranteed to return a number that is in the range [-y, y], whose square is smaller or equal to y and whose square of the absolute value increased of 1 is greater than y.

```
/*@ public normal_behavior
  @  requires y >= 0;
  @  ensures -y <= \result
  @   && \result <= y
  @   && \result * \result <= y
  @   && y < (Math.abs(\result) + 1)
  @        * (Math.abs(\result) + 1);
  @*/
public static int isqrt(int y){
  ...
}
```

Fig. 2. Sample heavyweight specification

2.2 Extended Static Checking (ESC)

While run-time checking code can certainly be useful for detecting and reporting errors, it leaves developers with the problem of handling them at run-time. Whenever possible, it would be more desirable if those errors could be prevented from occurring in the first place. Also, the earlier an error is found, the less expensive it is to correct. Static program checkers allow us to detect some of these run-time errors by static program analysis.

There exists a wide variety of static checking tools ranging from simple type checkers to formal program verifiers. Extended static checkers are a special kind of tool that, according to [11], generates verification conditions and logical formulas from a given program and passes them to an automatic theorem prover. Desirable characteristics of ESC tools are completeness and soundness i.e., catching all the errors, and triggering no false alarms (by reporting an error where there is none). On engineering grounds, such characteristics are not needed to benefit from ESC, especially since meeting such characteristics would imply over specification and reduced performances of the checkers [11][5].

ESC tools warn the user when there is an error or when the code does not implement its specification. By default, trivial specifications are assumed. A trivial specification implies that there are no special preconditions, that the execution may have any side effect, and that no specific post-condition should be expected. A default specification, if written explicitly, would look like Figure 3 [9].

A false alarm may indicate that the specification is too weak and needs to be strengthened. To do such a strengthening, programmers need to record the design decisions using the tool's annotation language that takes the form of a comment in the code. According to Flanagan et al. the annotation burden

```
/*@ behavior
  @   requires        true;
  @   assignable      \everything;
  @   ensures         true;
  @   signals         (Exception) true;
  @   ...
  @*/
```

Fig. 3. Default (i.e. implicit) method specification

coupled with the fact that un-annotated code generates an excessive quantity of warnings may lower the benefit/effort ratio below the acceptable boundary of the mainstream programmer [5].

2.3 ESC/Java2

ESC/Java is an ESC tool for Java that was developed subsequently to the ESC/Modula-3 tool. These two ESC tools cumulate more than 12 years of experience and have been successfully applied to tens of thousands of lines of code [11]. Even if ESC/Java uses a complex program verification technology, it looks like a simple type checker to a programmer. Moreover, since it performs modular checking (as opposed to whole-program checking) the level of complexity can be kept at a minimal [2]. ESC/Java2 features JML as its annotation language. Both heavy and lightweight specifications are accepted. Like its predecessor, ESC/Java2 is neither sound nor complete.

Coupling ESC with JML by using the phrases from a specification language as the ESC annotation language implies that the design decisions are recorded using a rich language and that such specifications can be reused, expanded, and passed to other JML tools-such as the runtime assertion checker compiler, JML RAC, and the formal program verification tool, LOOP [1]. Moreover, such an annotation language can take advantage of JML specifications of Java libraries in order to better reason about the code it is checking.

An important feature of JML is its adoption of a behavioral subtyping semantics [9] in which a method overridden in a subclass is required to preserve the contracts of its corresponding super class method [3]. In the next section we mention how we believe behavioral subtyping can be used to advantage.

2.4 Java Web-based Enterprise Application (WEA)

Most Enterprise Applications (EAs) involve a significant quantity of persistent data. Most often, databases are used to store and access that data. Another characteristic of EAs is the high number of user interaction screens. For

WEAs, Fowler recommends using the common three layered scheme, which isolates the domain logic (also referred to as business logic) from the presentation and data sources [6]. An interesting aspect of Java WEAs, is that most of them use the same small set of external libraries, super classes and interfaces for user interaction (e.g. *servlets.jar*) and database access (e.g. *java.sql.**). Due to the behavioral subtyping semantics of JML, we believe that in such a situation, the investment of specifying commonly used super classes and interfaces should reduce the effort required in specifying subclasses and lower the number of false alarms thus allowing developers to identify potential faults more easily.

3 Case study

3.1 Goals and approach

The main goal of our case study has been to assess the effectiveness of ESC/Java2 at helping developers increase software quality by detecting design and implementation faults. One of our early decisions was concerning the choice of application area. We opted for WEAs because it is one of the most active areas in Java development. We were also influenced by the availability of *SoenEA*, a small WEA framework that has been developed over the past two years for use in software architecture and design courses at Concordia University. *SoenEA* wraps code, such as the database connection and the servlet interface, so that students can use them easily without thorough knowledge of these technologies. The framework also comes with application samples.

Our study had two phases. During the first phase we applied ESC/Java2 to the *SoenEA* core and to sample applications that made use of the *SoenEA*. We will refer to this package as *A1*. The purpose of the first phase was to:

- gain experience in using ESC/Java2, while at the same time
- incrementally developing specifications for:
  · *A1* application
  · *SoenEA* core classes
  · *javax* and *java.sql* package modules used by *A1*.

ESC/Java2 performs modular checking [5][2] thus allowing us to work on one *A1* file at a time. This is very useful as ESC/Java2 initially reported hundreds of faults for *A1*. *A1* contained 2.1K ines of code (LOC), or 1.6K source lines of code (SLOC).

The purpose of phase two was to measure the reduction in false alarms due to the use of the annotated versions of *SoenEA* core, *javax*, *java.sql* modules developed in phase 1. In addition to the *A1* package we analyzed two more applications totaling 7.6K LOC (or 5K SLOC). We will refer to the latter as the *A3* package.

3.2 General results

These two phases required approximately 4 person-weeks of effort (full-time). At the conclusion of phase I we had created:

- An annotated version of the *A1* package (i.e. *SoenEA* core and the *A1* application). Overall this represented an increase in size by approximately 4% due to the annotations (i.e. about 100 LOC).
- Lightweight specifications for the *javax* and *java.sql* package modules — 90 SLOC (379 LOC). These lightweight specifications consisted mostly of annotations about the class attributes and method arguments restricting them to being non-null.

In phase II, use of the created specifications reduced false alarms by 9% on average for the *A1* and *A3* packages as compared to use of ESC/Java2 without the specifications.

In the subsections that follow we relate some of our experiences in specifying the external libraries (*javax* and *java.sql*) as well as the SoenEA core. The material presents issues of increasing complexity (to emulate, to some extent, the order in which we usually had to deal with them). The final subsection covers the most interesting faults reported by ESC/Java2 for the *A1* package.

Before discussing the number of faults uncovered by ESC/Java2 it becomes essential at this point to provide our definition of “fault”. By “fault”, we refer to something that is wrong with respect to the intended use of a method (or in general, a class). The intended use of a method is what is recorded in its specification. When there is no explicit documented specification, a default implicit specification is assumed. A “fault” occurs when the code does not satisfy the specification. According to DBC this can occur either because a client calls a method when the method’s pre-condition is not satisfied or when a method returns and its post-condition is not satisfied. A false alarm due to a missing explicit specification will be recognized as a fault and named a “specification fault”. False alarms over external libraries denote missing specifications in ESC/Java2, whereas false alarms over user modules denote true specification faults.

3.3 Specifying javax, java.sql and the SoenEA core

ESC/Java2 performs static checking and reasoning using the JML annotations present in the code or specification files. In the absence of a specification (as

```
soenEA.applications.assignment.a3.ts.TaskTDG: findAll() ...

------------------------------------------------------------
.\soenEA\applications\assignment\a3\ts\TaskTDG.java:35:
  Warning: Possible null dereference (Null)
                return dbStatement.executeQuery();
                                  ^
------------------------------------------------------------
    [0.19 s 10054064 bytes]  failed
```

Fig. 4. Sample false alarm / specification fault

```
public class TaskTDG {
  ...
  public static ResultSet findAll()
  throws Exception, SQLException {
    Connection db = DbRegistry.getDbConnection();
    PreparedStatement dbStatement =
      db.prepareStatement("SELECT * from " + TABLE_NAME);
    return dbStatement.executeQuery();      // line 35
  }
  ...
}
```

Fig. 5. TaskTDG code excerpt

would be the case for a method of the *javax* class at the start of phase I) ESC/Java2 assumes the default specification (Section 2.2) which is very permissive and seldom satisfactory. This implies that in the absence of specification, ESC/Java2 may trigger warnings, many of which would be false alarms or specification faults. (Thankfully ESC/Java2 comes with JML specifications for the most common Java classes, which, e.g. prevents it from warning about possible null dereferences in the case of a simple *System.out*.)

An example of a specification fault that ESC/Java2 reported early in phase I is given in Figure 4. The corresponding code is given in Figure 5. ESC/Java2 reports that the variable *dbStatement* could be null at the indicated point in line 35. This is not the case since this variable is set to the return value of *db.preparedStatement()* which always returns a reference to an object [15]. This is a false alarm due to a lack of explicit specification of *preparedStatement()*.

Fixing this false alarm is simple: we create an explicit specification for the method stating that it does not return a null result — see Figure 6. Most of

```
package java.sql;

public interface Connection {

  //@ ensures \result != null;
  public PreparedStatement prepareStatement(String sql)
  throws SQLException;

}
```

Fig. 6. Sample lightweight spec created for Connection.preparedStatement()

the specifications created for *javax* and *java.sql* modules where of this nature: i.e. specifying that arguments and or returned results would not be null. ESC/Java2 is capable of static checking of far more properties but statically detecting all possible null dereferences was deemed sufficient for this iteration of the case study.

Like the specifications of the external libraries, the majority of the annotations added to the SoenEA core were of the same lightweight nature. This allowed us to uncover some very interesting faults nonetheless, which we report in the next section.

4 Specific design and implementation faults

In this section we report on the most interesting faults uncovered by ESC/Java2 in the *A1* package during the second phase of our study. ESC/Java2 reported 102 faults for the *A1* package. Of the 102 faults that were identified, 90% were specification faults, i.e. they were due to a lack of design documentation/specifications. Such faults are eliminated by writing design documentation in the form of method API specifications. The faults presented next are taken from the remaining 10% of the faults.

4.1 Incompletely propagated design changes

At least two of the faults were manifestations of incompletely propagated design changes. For example, one of the domain logic classes representing a *Task* in the *A1* application underwent the following design change: initially the class fields were initialized by means of setters; subsequently it was decided that all fields were to be initialized by means of the (non-default) class constructors. A consequence of this design change is that class constructor arguments of reference types could no longer be null. Such a fact had been properly documented in the *Task* class but not all calls of the constructors

```
public class User { ...
  private long id;
  private String loginId;
  private String name;
  private List originatedTasks;
  private String password;

  public boolean equals(Object obj) {
    if(obj == null || !(obj instanceof User))
    return false;
    User other = (User) obj;
    return this.id == other.id &&
           this.loginId.equals(other.loginId) &&
           this.name.equals(other.name) &&
           this.password.equals(other.password);
  }
}
```

Fig. 7. User class excerpt

respected these constraints either because:

- not all client classes were appropriately updated, or, as often happens,
- in subsequent updates to the application, developers were still relying on the old API semantics.

These two scenarios demonstrate the advantage of formally documented design decisions and the use of extended static checking.

Another similar example occurred in the *SoenEA* database registry. An initial version of the class tried to enforce that a connection field would never be null. However, it was realized that this was infeasible. It is interesting to note that had ESC/Java2 been used earlier, this design error would have been uncovered from the start. The class was redesigned, but ESC/Java2 helped uncover some situations in which designers had not anticipated that the connection field could still be null.

4.2 Possible violation of behavioral subtyping

In the *User* class a method named *equals* was defined (Figure 7), thus overriding *Object*'s *equals* method. ESC/Java2 reported that the superclass specification (i.e. the specification of *Object.equals()*) might not hold for *User.equals*. It was initially thought that the cause of this fault was that *User.equals* only compared four of its five attributes. However, after discussion with designers,

```
soenEA.general.app.Servlet: forwardAbsolute(java.lang.String,
 javax.servlet.http.HttpServletRequest,
 javax.servlet.http.HttpServletResponse) ...
-----------------------------------------------------------
soenEA/general/app/Servlet.java:109:
 Warning: Possible null dereference (Null)
        dispatcher.forward(request, response);
                  ^
```

Fig. 8. Null dispatcher error

```
public void forwardAbsolute(String target,
HttpServletRequest request, HttpServletResponse response)
throws ServletException, java.io.IOException
{
    RequestDispatcher dispatcher =
getServletContext().getRequestDispatcher(target);
    dispatcher.forward(request, response);
}
```

Fig. 9. Servlet class excerpt

it was realized that this was deliberate. Hence, in order to avoid confusion, it was decided that the proper corrective action was to rename *User.equals* to *User.similar*. This example illustrates the power of behavioral subtyping: carefully written super class specifications allow ESC/Java2 to uncover semantic errors.

4.3 Unchecked dispatcher

ESC/Java2 detected a fault in the *SoenEA* core that, we have come to know, is apparently common for novice WEA developers (Figure 8). The framework assumed that the servlet dispatcher was always initialized. This is not the case [14]. ESC/Java2 provided a warning to this effect (based on default implicit specifications of *getRequestDispatcher*, which in this case was correct). The faulty code is given in Figure 9.

4.4 Missed exceptional condition

Database connection information (e.g. login id, password) is recorded in a property file that the *SoenEA* framework reads. While the case where the file is not present was properly handled, the case where the file did not contain the required properties was not. This was not caught during code review nor during testing.

4.5 Other faults

A few of the remaining faults were simple possible null pointer dereferences. An example of this occurred in a file that was recently added to the framework, which had not been tested carefully. ESC/Java2 detected three possible null pointer dereferences in it. Only an extensive use of the framework would have been able to catch these, since the class was an exception adapter that would only be used under exceptional circumstances.

5 Conclusion

We believe writing JML specifications and verifying the match between them and the code with ESC/Java2 was a very helpful exercise. The time spent during this case study has allowed us not only to identify and correct design issues, but it has also forced us to think about and document the design decisions that had previously been made. Moreover, this documentation can be automatically verified by the tool. We have raised our level of trust in the framework and, through the enforcement of behavioral subtyping, feel the framework is a better candidate for reuse and expansion.

In this case study, ESC/Java2 proved itself to be useful for WEAs. However, it is a general purpose tool that is not limited to a specific domain of application. The use of ESC/Java2 is likely to spread to other domains of application. To our knowledge, this case study was the first one involving ESC/Java2 and the WEA domain. Perhaps a drawback of JML in that particular domain is that is does not currently support reasoning about concurrency. However, a proposal has just been published to address this issue [13]. Nonetheless the lightweight JML specifications we had to write for *javax* and *java.sql* can be reused or supplemented by the JML community and can contribute to making JML and its supporting tools, like ESC/Java2, more convenient to use for Java developers.

Our case study involved a relatively small framework and application, but since it performs modular checking, we believe that ESC/Java2 will scale up to larger applications. For this case study, we used the latest development release of ESC/Java2 (December 2004). It is considered to be in a late alpha cycle. As such we did encounter errors with the tool but none that could not be worked around. In fact the lead developers were very responsive to our problem reports and requests for assistance when the tool failed.

Of course there is a cost associated with the creation of specifications. As this was our first case study involving ESC/Java2, an important part of our effort was dedicated to learning about the tool and about the particular form of JML specifications that best suite it. Several false alarms were also due to missing specifications of *javax* and *java.sql* classes. Moreover, as of now, ESC/Java2 is not integrated into any IDE. We are confident that as research progresses and ESC/Java2 becomes more mature and easy to use, the cost/benefit ratio will become more and more appealing.

6 Future work

Now that some of the *javax* and *java.sql* classes have been specified, in the future, we would like to perform this study again with other WEAs and see if the annotation burden has been lowered in a significant manner. Since every year the *SoenEA* framework is used by more than a hundred students, we plan to provide them with the annotated version of the framework and verify whether they think using ESC/Java2 can improve the quality of their WEAs and help them identify and correct faults. Yet another interesting avenue would be to reuse the annotated *SoenEA* framework, applying other JML compatible tools to it, and then checking how much the written specifications can be reused and what benefits WEA developers can get out of them.

7 Acknowledgement

The authors would like to thank Daniel Sinnig, Stuart Thiel, and the anonymous reviewers for their valuable feedback as well David Cok and Joseph Kiniry for helping us with ESC/Java2 and answering all our questions.

References

[1] Burdy, L., Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino and E. Poll, *An overview of JML tools and applications*, International Journal on Software Tools for Technology Transfer (STTT) (2004), to appear.
URL ftp://ftp.cs.iastate.edu/pub/leavens/JML/sttt04.pdf

[2] Cok, D. R. and J. R. Kiniry, *Esc/java2: Uniting esc/java and jml*, in: *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices* (2004), pp. 108–128.

[3] Dhara, K. K. and G. T. Leavens, *Forcing behavioral subtyping through specification inheritance*, Technical Report 95-20c, Department of Computer Science, Iowa State University, Ames, Iowa, 50011 (1995).
URL ftp://ftp.cs.iastate.edu/pub/techreports/TR95-20/TR.ps.gz

[4] *Esc/java2*.
URL www.sos.cs.ru.nl/research/escjava/

[5] Flanagan, C., K. R. M. Leino, L. Lillibridge, G. Nelson, J. B. Saxe and R. Stata, *Extended static checking for Java*, ACM SIGPLAN Notices **37** (2002), pp. 234–245.

[6] Fowler, M., "Patterns of Enterprise Application Architecture," Addison Wesley, 2002.

[7] Johnson, S., *Lint, a c program checker* (1978).
URL citeseer.ist.psu.edu/johnson78lint.html

[8] Leavens, G. and Y. Cheon, *Design by contract with jml* (2003).
URL `citeseer.lcs.mit.edu/leavens03design.html`

[9] Leavens, G. T., A. L. Baker and C. Ruby, *Preliminary design of JML: A behavioral interface specification language for Java*, Technical Report 98-06y, Iowa State University, Department of Computer Science (2004), see `www.jmlspecs.org`.
URL `ftp://ftp.cs.iastate.edu/pub/techreports/TR98-06/TR.ps.gz`

[10] Leavens, G. T., K. R. M. Leino, E. Poll, C. Ruby and B. Jacobs, *JML: notations and tools supporting detailed design in Java*, in: *OOPSLA 2000 Companion, Minneapolis, Minnesota*, ACM, 2000, pp. 105–106.
URL `ftp://ftp.cs.iastate.edu/pub/techreports/TR00-15/TR.ps.gz`

[11] Leino, K. R. M., *Extended static checking: A ten-year perspective*, Lecture Notes in Computer Science **2000** (2001), pp. 157–??
URL `http://link.springer-ny.com/link/service/series/0558/bibs/2000/20000157.htm`

[12] Meyer, B., "Object-Oriented Software Construction," Prentice-Hall, Englewood Cliffs, 1997, second edition.
URL `http://www.prenhall.com/allbooks/ptr_0136291554.html`

[13] Rodríguez, E., M. B. Dwyer, C. Flanagan, J. Hatcliff, G. T. Leavens and Robby, *Extending sequential specification techniques for modular specification and verification of multi-threaded programs*, Technical Report SAnToS-TR2004-10, Kansas State University, Department of Computing and Information Sciences (2004), to appear in *ECOOP 2005*.
URL `http://spex.projects.cis.ksu.edu/papers/SAnToS-TR2004-10.pdf`

[14] *Sun java servlet api documentation*.
URL `java.sun.com/j2ee/1.4/docs/api/javax/servlet/package-summary.html`

[15] *Sun java sql api documentation*.
URL `java.sun.com/j2se/1.4.2/docs/api/java/sql/package-summary.html`

[16] *Quarterly retail e-commerce sales 3rd quarter 2004* (2004).
URL `www.census.gov/mrts/www/data/html/3q2004.html`