# Error Detection

Klaus Renzel

sd&m – software design & management GmbH & Co. KG
Project ARCUS[1], Thomas-Dehler-Str. 27 – D 81737 München, Germany
Email: Klaus.Renzel@sdm.de, Phone: +49-89-63812-251

**Abstract**

Before we can handle an error or failure we have to detect it. Error Detection shows how to install error detectors within your software: it enriches the code with a number of run-time checks. Those checks are a prerequisite for handling software faults successfully and to avoid system crashes.

**Also Known As**

Error Traps, Assertion Checking

**Context**

Reliability is a major requirement for your software system. Unfortunately, no software system can be assumed to behave totally correct. Even careful testing, debugging or the use of formal methods does not guarantee fault-free software. Thus, your system must expect errors or failures at run-time and handle them by appropriate mechanisms.

**Example**

Imagine a simple library system. An abstract specification for this system may introduce datatypes `User`, `Book` and `Library`. A function `LendBook` specifies the user transaction of borrowing a book from the library. The number of books which can be borrowed by a single user is restricted to a certain limit modeled by a constant `BorrowingLimit`. A predicate `Authorized` characterizes whether a user is allowed to borrow a book or not (e.g. the system distinguish between different user classes).

```
METHOD LendBook (User u,  Book b)
BEGIN
      -- Possible error situations:
      --    * The parameters are not well intialized
      --    * The book is not part of the library
      --    * The user is not known or authorized
      --    * The user will exceed the limit
      --    * The user already borrowed this book
      --    * The book is currently lent to another user
      ...
END
```

[1] This work is sponsored by the German Ministry of Research and Technology under project name ENTSTAND and is part of a larger effort to collect patterns for business information systems, currently under development by the ARCUS team (http://www.sdm.de/g/arcus).

For an implementation the error behaviour is important: How to implement the operations for the library system to ensure that possible errors such as a violation of the borrowing constraint are detected as early as possible? Think of an implementation of the method `LendBook`:

How and where to check for error conditions within this method?

**Problem**

How to detect errors or failures at run-time to enable the system to react to such situations properly?

**Forces**

- *Specification*: A failure can only be detected in relation to a specification of the correct behaviour. The specification itself is assumed to be correct. In this respect, error detection means to verify the state and behaviour of a system against the specification at run-time.

- *Completeness versus simplicity*: Often it is not feasible and even sensible to check an implementation completely against its specification. Either the implementation of an operation is too simple or checking code is at least as complex as the implementation itself.

- *Complexity versus criticality*: Balance necessary overhead (additional complexity, code size of source and executable) with the severity and frequency of errors.

- *Performance versus reliability*: On the one hand, we want minimal performance penalties, and, on the other hand, we want to be able to detect nearly all kinds of errors as soon as possible.

- *Robustness and consistency*: It is desirable to automate error checking as much as possible because automation supports a coherent design and correct implementation.

- *Maintainability*: To preserve maintainability of the application code, you should avoid cluttering the code with a mass of error detectors.

- *Flexibility*: To allow more flexibility, you should provide a way to activate and deactivate the error detectors.

- *Logging*: You should provide a basic error log capability into which error detection events are recorded.

**Solution**

Specify the behavior of the classes and methods precisely using invariants, pre- and postconditions [Mey88].[2] Instrument methods (except inline methods and macros) to check for those conditions as can be done with reasonable effort. The table shows the most common checks for a method m.

The pre- and postconditions are constraints about the class's internal state and the environment's state as formulated in the specification. A developer may insert additional checks for special assertions, such as loop invariants. Sometimes it is necessary to think about alternative conditions to be checked because the specified ones are not practicable.

---

[2] Every constraint must be made explicit (e.g. to avoid reuse errors [JM97]). To enforce a particular constraint (e.g. a constraint on the relation between a group of objects) can be a difficult design task.

| What? | Where? |
|---|---|
| Invalid parameters. | On entry of the method. |
| Violation of the pre-condition. | On entry of the method. |
| Unexpected results or failures of methods called by m. | Immediately after return of the method. If the language supports exception handling signaling of an exception automatically invokes an appropriate exception handler. Thus we do not need to instrument the code with an additional check for the result and the exception handler code can be well separated from the application code. |
| Violation of a method's invariant. | We have to distinguish between two kinds of invariants: some invariants are related to the whole class and they are checked at the end of every method of that particular class. This works well (the invariant holds on entry of every method) as long as the data of the class is only accessed and manipulated by these methods. Checking an invariant on entry as well as on termination of a method is necessary if an invariant concerns properties of shared classes (aliasing) or if an invariant is restricted to particular methods. In the latter case, the invariant can also be expressed as part of the pre- and postcondition. |
| Violation of the post-condition. | On termination of the method. |

**Structure**

The following pseudo-code illustrates the structure of a method instrumented for error detection. We use the notation **[** *assertion* **?** *action if the assertion is violated* **]** for a general error detector.

```
METHOD AnyMethod(aType1 aParam1, aType2 aParam2, …) : aReturnType
BEGIN
      ------- error detection header ------------------------
      [ aParam1 valid ? raise exception for invalid parameter ];
      [ aParam2 valid ? raise exception for invalid parameter ];
      [ invariant holds ? raise exception for violated invariant ];
      [ precondition holds ? raise exception for violated precondition ];

      -------- normal method body ----------------------------
      …
      -- do something
      [ special test ? raise exception ];

      Result = aClass.OtherMethod(aValue);
      [ exptected Result ? raise exception ];
      …
      -------- error detection footer --------------------------
      [ invariant holds ? raise exception for violated invariant ];
      [ postcondition holds ? raise exception for violated postondition ];
      [ return value valid ? raise exception for invalid result value];
      RETURN aValue;

HANDLE
```

```
      handle exceptions raised within the block
END
```

**Example Resolved**

The specification of the library system may be given by (using an adhoc notation)

```
SPECIFICATION CLASS LibrarySystem
BASED ON Archive, User, Book, ...
INVARIANTS
      ∀ u ∈ GetUsers() . NrOfElems(BorrowedBooks(u)) ≤  BorrowingLimit()
METHOD LendBook (User u,  Book b)
REQUIRES
      u.IsValid() ∧ b.IsValid() ∧ b ∈ GetArchive() ∧ u ∈ GetUsers() ∧
      Authorized(u,b) ∧ ¬( ∃ u': User. u' ∈ GetUsers() ∧ b ∈ BorrowedBooks(u'))
ENSURES
      b ∈ BorrowedBooks(u) ∧
      NrOfElems(BorrowedBooks(u)) = NrOfElems(old BorrowedBooks(u)) + 1

METHOD BorrowingLimit : Nat    ...
METHOD GetUsers : Set(User)    ...
METHOD GetArchive : Archive    ...
METHOD BorrowedBooks(User u) : Set(Book)      ...
METHOD Authorized(User u, Book b) : Bool      ...
...
```

The implementation makes use of a class `BookAccount`. For every library user exists an account to administer the books lent to him.

```
METHOD LendBook(User u,  Book b)
BEGIN
      ------- error detection header ------------------------
      [ u.IsValid()            ? raise InvalidParameter ];
      [ b.IsValid()            ? raise InvalidParameter ];
      [ b ∈ GetArchive()       ? raise UnknownBook ];
      [ u ∈ GetUsers()         ? raise UnknownUser ];
      [ Authorized(u,b)        ? raise NoAccess ];
      [¬b.IsBorrowed()         ? raise BookInUse ];
      [ CheckLimit(u)          ? raise UserExceedsLimit ];
      -------- normal method body ----------------------------
      ...
      b.MarkAsBorrowed();
      GetBookAccount(u).EnterBook(b);
      ...
      -------- error detection footer --------------------------
      [ CheckLimit(u)          ? raise UserExceedsLimit ];
      [ b ∈ BorrowedBooks(u)   ? raise IncorrectLending ];
HANDLE
      handle exceptions raised within the block
END
```

The implementation shows that we use an extra method `CheckLimit` to implement the class invariant. `IsValid` is also a special check method provided by the datatypes to examine the definedness of values. The implementation leaves out the last assertion in the specification. Because of the special

keyword **old** which yields the value of a variable before method execution, a possible implementation has to store this value in some temporal variable, for example:

```
Int tmp_nrbooks = NrOfElems(BorrowedBooks(u));
-------- normal method body ----------------------------
...
-------- error detection footer -------------------------
[ NrOfElems(BorrowedBooks(u)) = tmp_nrbooks + 1 ? raise InvalidNrOfBooks ];
```

**Consequences**

- *Specification*: The implementation of the detectors is guided by the specification. As already mentioned we must assume that the specification is correct, so this solution is not helpful to detect errors in the design specification. Whether this solution is really effective and can detect a huge number of errors also depends on the quality (completeness) of the specification. If the specification carefully exposes the pre- and postconditions and invariants, a correct implementation according to this pattern will detect most implementation errors. The solution is not suited to detect non-termination (e.g. endless loops).

- *Complexity versus criticality*: Whether this solution detects errors as early as possible depends on the method's size. The smaller the methods the higher the frequency of checking and thus detection is closer to the original cause of an error. The size of a method varies depending on the programming style and language. Roughly speaking methods in object oriented languages tend to be smaller than procedures in imperative languages.

- *Performance vs. reliability*: The solution enriches the code of nearly every method. On the one hand this enables early detection of errors and possibly prevents small errors to become larger problems. On the other hand it strongly effects the performance of an application. Of course the kind of implementation influences the performance, but to really speed up the only choice is to switch off error detection. A compromise would be to restrict error detection to the critical parts of an application. Whether you restrict assertion checking in a release version due to performance depends on the application type. For example, in the control software of Ariane 5 [JM97] developers remove some checks to gain performance, whereas in our practice at sd&m performance of business information systems seldom essentially suffers from checking code but mostly from database related issues.

- *Robustness and consistency*: A critical situation is the incorrect implementation of an error detector itself. This danger is increased by the fact that pre- and postconditions as well as invariants are mostly hand-coded. For example, the implementation of assertions may introduce loops.

- *Maintainability*: As most of the checks are done when entering or leaving a method they can be well separated from the functionality within a method's body. Other aspects relevant to maintenance depend on the implementation (e.g. use of special check methods, collecting and logging information describing the error context).

- *Flexibility*: The checking of preconditions aims at detecting errors within the clients of a method, but the actual checking is done by the callee. Therefore, separate compilation of the caller and callee's code reveals some inflexibility. When compiling callee's code we already have to decide whether we want to switch off or on assertion checking (by use of a single flag). Instead, we want to decide about checking preconditions when compiling the clients code. Checking on the client side also provides more optimization possibilities for compiling assertions [GK97].