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].

- *Logging*: An error detector can write valuable information to a log file. As detection is close to the originating event the information about the system's state at this point is very important for later analysis of the error situation. Thus, the solution supports debugging of programs, especially when interactive debugging is not possible. The disadvantages are: the need of every detector to access a module offering logging services and the negative effect of logging operations on system performance.

Implementation

- *Check methods:* By introducing additional methods to a class for checking particular properties (e.g. an invariant, the consistency of a class) we can avoid redundant code in a number of detectors which have to verify these properties. In the library example we use methods `IsValid` and `CheckLimit`.

- *Optimizations:* It is not necessary to check assertions and invariants within every method and even for a method it may depend on the caller of the method. For example, it is not practicable to instrument simple get-methods to verify that they do not modify the object's current state. In the library example the checking code calls other methods. It is desirable that assertion checking within those methods is only triggered when they are called from other objects. The same applies to internal (private) method calls.

- *Detector actions:* Once an error is detected a number of actions should be triggered: collection of context information, creation of an exception object, reporting the exception to the error log, cleaning up resources, trying to reach a consistent state and finally signaling the exception to the caller. Again, to avoid redundant code it is necessary to implement macros or methods for these actions.

- *Activation, Deactivation:* Introducing different checking levels and switches offers more flexibility concerning the intensity of checking. C and C++ programmers often use preprocessor directives (like `#define`, `#ifdef`) to implement compile-time switches (either a global switch via makefile or locally within the source files), which allows the complete removal of error detectors from the code. For a more fine-grained control, it is necessary to distinguish between different kinds of checks or to provide switches on a per class level. Consider who should be able to activate and deactivate detectors: is it sufficient to fix the error detection mode at compile-time by the software developers or do system administrators need run-time configuration capabilities?

- *Macros:* To implement error detectors with macros helps to keep the code attachments small and readable, they prevent redundant code, are very flexible, easy to change or remove, compile time overhead is acceptable and run-time performance is very good. Note that it is also possible to use a preprocessor like that for C and C++ for other languages (e.g. Java, Cobol). Macros, however, have drawbacks for debugging as you cannot step into a macro with a debugger and macro expansion increases the code size.

- *Generation:* Generally, it is helpful to generate as much code from the specification as possible. How much code for error detection might be automatically derived from the specification depends on the specification language, on the one hand, and the programming language, on the other hand. But it is always possible to generate code templates, which must be completed by hand-coding. Especially, the precondition and invariants are often specified by natural language, which makes it impossible to generate code for them. Otherwise, specification languages which support precise specification of preconditions and invariants by logic formulas (predicate calculus) also require to refine these constraints as long as they are executable. Eiffel is a programming language which includes directives for executable specifications.

- *Multiple Return Statements*: The verification of assertions becomes more difficult in the case of multiple return statements, because assertion checking must be done on different places within a method. Different solutions are possible: We can forbid multiple return statements, we can use some compile environment which automates multiple code instrumentation or we can create a special check object triggering the assertion checking via the automatic destruction of the object.

- *Inheritance of assertions*: Use of inheritance as behavioural subtyping requires to keep or weaken a precondition and to keep or strengthen a postcondition. Advanced implementation of assertions consider inheritance and support the checking of derived classes by some automatism.

- *Old Values*: Often postconditions express state changes of a method by use of variable's values before method execution („old values"). In the implementation we have to save the old values temporarily to make them accessible while checking postconditions. For example, the Eiffel language supports access to old values by an extra operator `old`, which can be used in the specification of postconditions.

Sample Code (C++)

As the exception log should contain some useful maintenance information, you need to include lots of parameters into an exception's constructor. Most of these parameters such as `__LINE__` numbers or `__FILE__` names may be obtained automatically or may be generated using function calls like `_actualTime()`. Writing all these parameters by hand is far too expensive.

In C++ we can use parameterized macros that contain the minimum possible number of actual parameters. We can obtain all other information using macros like `__LINE__`, `__FILE__`, `__FUNCTION__` or whatever your development environment supports.

The actual macros you use depend on your project's requirements and programming environment. The following macros give an impression of what has been used successfully in various sd&m projects:

```
// define detectors
#define AssertTemplate(CONDITION,EXID,TEXT)                                \
 if ( !(CONDITION) )                                                       \
      throw ExAssertionFailure(#EXID,#CONDITION,TEXT,__FUNCTION__,         \
      _LINE__,__FILE__,__DLL_NAME__,__EXE_NAME__,__EXTIME__,               \
      __USER_NAME__)

#define AssertParam(CONDITION, TEXT)                                       \
   AssertTemplate(CONDITION, EX_ILLEGAL_PARAM, TEXT)

#define AssertPrecond(CONDITION, TEXT)                                     \
   AssertTemplate(CONDITION, EX_VIOLATED_PRECONDITION, TEXT)

#define AssertInvariant(CONDITION, TEXT)                                   \
   AssertTemplate(CONDITION, EX_VIOLATED_INVARIANT, TEXT)

#define AssertPostCond(CONDITION, TEXT)                                    \
   AssertTemplate(CONDITION, EX_VIOLATED_POSTCONDITION, TEXT)
```

Other approaches in C++ use include files to insert the necessary code for error detection, implement check methods by templates, or use inline methods. The listed approaches can also be combined. Be

careful using __LINE__ and __FILE__ macros within check methods. If they are expanded, the preprocessor will show you the source line of the check method and not the line in which the template was used. It is an advantage of the macro approach that we can hide __LINE__ and __FILE__ within the assert macros. Otherwise, every call of a check method within the application code must explicitly pass __LINE__ and __FILE__ as parameters.

Sample Code (Cobol)

We took the following examples from the error handling of sd&m's **TLR** project [TLR95], which uses a standardized mechanism to perform result checks and type checking of variables. The following code excerpt (in an extended Cobol language) for a module operation illustrates the mechanism:

```
*****************************************************************
OPERATION DoSomething
*****************************************************************
PARAMETER
      in aParamID : aType
...
BEGIN
%CHECK SF (aParamID of $PARAMETER, aType) // checking the type of the parameter
...
%CALL ModuleName OtherOperation
       ( // parameter values...
         aKey )

%CHECK-RC (RC-OK, RC-NOK)
if $GRC = RC-NOK then
      %SET-DF (anErrorNumber, aKey)
end-if
...
%RETURN(RC-OK)
END-OPERATION
```

%CHECK is the type-checking command and %CHECK-RC compares a list of expected return-codes with the actual return-code. Deviation of the latter from the expected return-codes results in a system error.

From this high-level Cobol code, pure Cobol is generated. Every module needs some types and variables for exception handling which are implemented by the following data structures:

```
* ------------------------------ types ----------------------------------
01   type-constants
      ...
      05 rc-global                    .
         10 RC-OK                     pic x(25)  value 'RC-OK'.
         10 RC-NOK                    pic x(25)  value 'RC-NOK'.
         ...
      05 module-state                 .
         10 NORMAL-STATE              pic x(2)        value 'OK'.
         10 SE-STATE                  pic x(2)   value 'SE'.
         ...
      05 exception-id                 .
         10 SE-PARAMETER              pic x(25)  value 'SE-PARAMETER'.
         10 SE-UNEXPECTED-RC          pic x(2)   value 'SE-UNEXPECTED-RC'.
         10 SE-PRECONDITION           pic x(2)   value 'SE-PRECONDITION'.
         ...
```

```
* ---------------------- internal module variables ----------------------
 01   internal-variables
      ...
      05 trace-buffer                .
           ...
      05 format-string         pic x(80).
           ...
* ------------------------- global variables ----------------------------
 01   global-variables
      ...
      05 rc-global                    pic x(25).
      05 module-state          pic x(1).
      05 current-operation            pic x(25).
      05 exception-id          pic x(25).
      05 exception-loc         pic 9(3).
      ...
```

The type check command yields the following code:

```
* --- check type of aParamID ---
*
evaluate aParamID of ...
when ... continue
when ... continue
when other
      move aParamID ...
      move RC-NOK to rc-global of global-variables
*           >>> System Error - Location 4 <<<
            move SE-STATE of module-state of type-constants
                  to module-state of global-variables
            move 4 to exception-loc of global-variables
            move SE-PARAMETER to exception-id of global-variables
            move ...
            perform handle-exception
*           >>> End of System Error - Location x <<<
end-evaluate
```

If the parameter value does not match the type constraints, a system error is produced. The `module-state` switches to `SE-STATE` (system error state) and an automatically generated sequence number (4 in this case) serves as an identifier for this error code. The identifier is written to the error log and helps to navigate back to the corresponding location in the source code. The constant `SE-PARAMETER` (defined within the data structures) describes the type of the error and is assigned to the variable `exception-id`. All the information is used by the routine `handle-exception` which finally writes the error log. `handle-exception` is a common routine of the exception handling component and is included in every generated Cobol module.

The command `%CHECK-RC(RC-OK, RC-NOK)` similarly expands to:

```
if not rc-global of global-variables = RC-OK and
   not rc-global of global-variables = RC-NOK
then
*     >>> System Error - Location 7 <<<
      ...
      move SE-STATE of module-state of type-constants
            to module-state of global-variables
```

```
      move 7 to exception-loc of global-variables
      move SE-UNEXPECTED-RC to exception-id of global-variables
      move rc-global of global-variables
            to trace-xparam of trc-buf-filed of internal-variables (1)
      move length of rc-global of global-variables
            to trace-length of trace-buffer of internal-variables (1)
      move ´The return-code is: %s` to format-string of internal-variables
      perform handle-exception
*     >>> End of System Error - Location 7 <<<
end-if
```

Sample Code (Smalltalk)

The **DaRT** project at sd&m uses an assertion class (`FraAssert`[3]) in Smalltalk which provides a method `that: with:` to check a condition. For example

```
Assert that: ['Parameter is defined'] with: [aParam isDefined]
```

checks a parameter. The receiver(`Assert`)of the message is a global variable. The code behind the method can be switched by a class method `off`, which substitutes the class assigned to the variable `Assert` by a subclass (`FraNoAssert`) with an empty implementation. This offers the flexibility to adjust the assertion checking dynamically.

Of course, in the client class remains a method call without any functionality which produces a run-time penalty. The following code shows the implementation of the class `FraAssert`:

```
FraSysDomainObject subclass: #FraAssert
...
that: aStringOrBlock with: aBlock
     aBlock value == true
          ifFalse: [ | tmpString |
               tmpString := aStringOrBlock isString
                    ifTrue: [aStringOrBlock]
                    ifFalse: [aStringOrBlock value].
               (self app msg: #Assertion) arg: tmpString; raiseSysError
          ]

!FraAssert class methodsFor: 'class initialization'!
initialize
     self on

!FraAssert class methodsFor: 'toggle'!
off
     Smalltalk at: #Assert put: FraNoAssert

on
     Smalltalk at: #Assert put: FraAssert
```

In case of an assertion violation, the method `that: with:` creates a new assertion message object (by use of the message identifier #`Assertion`). The first parameter of the method, which offers a

[3] The class is part of the *Frammento* framework which was developed within the DaRT project. All class names in the framework have a prefix `Fra`.

description of the assertion, is passed to the message object as an argument. Finally, the exception is raised by the message `raiseSysError`. The code for `FraNoAssert` is straightforward:

```
FraAssert subclass: #FraNoAssert
...
```

```
!FraNoAssert class methodsFor: 'assertion'!

that: aString with: aBlock
     "Nothing is checked."
```

Known Uses

Assertion checking is a topic since the beginnings of structured programming (languages). Nevertheless, there still exists little support for executing assertions in popular programming languages (e.g. Java) and their compilers.

This pattern is mainly influenced by the features of the **Eiffel** language [Mey88]. Eiffel contains `require` and `ensure` clauses to describe pre- and postconditions, supports the specification of invariants and also offers a `check` command to formulate assertions. To what extent these commands are executed at run-time can be configured within the compile environment. In the extreme case they are just documentations.

Another language supporting executable assertions is **Sather** [Sather] developed at the University of Berkeley. Similar to Eiffel the syntax allows pre- and postconditions, invariants, and general assertions. Invariants are placed in a method with standardized name which is called before and after every public method. In contrast to Eiffel, Sather not assists in inheritance of pre- and postconditions.

The **DaRT** project at sd&m uses assertions and implements a general assert method as part of their *Frammento* framework, as shown in the sample code section.

Most C++ projects we know use special assert macros. For example, **BTS** [Kun96] use the assert macro provided by MVC++. Because this macro is compiled in the debug version only, it is used solely for errors which can be detected while testing. There are other examples in the **DATEV**, **HYPO** and **EASY** projects. Some distinguish between an `AssertDebug` macro specialized for debug mode and a general `Assert` macro for checking conditions during development as well as operation.

Nana [Nana] is a library for annotating C or C++ programs using GNU's cpp and gdb. It supports postconditions referring to the before state of an operation and contains macros for existential and universal quantification (over finite collections).

As already mentioned the **TLR** project implements standardized error detection features into their sophisticated Cobol development environment. This environment is also used by and adapted to a number of other sd&m projects.

Further Reading

For detection of memory errors look for available tools on the market if the programming language or the environment does not support resource management very well. The book by D. A. Spuler [Spu94] contains useful tips and techniques for instrumentation of C and C++ code.

[LK86] discusses many specification and implementation aspects, such as assertions and defensive programming, in the context of the CLU language.

A recent research paper [GK97] deals with language and compiler impacts of executable assertions.

Acknowledgments

I would like to thank Jens Coldewey, Wolfgang Keller, Falk Carl, Gerhard Albers, Chad Smith, and last but not least my shepherd Robert S. Hanmer for their support.

References

[GK97] **K. J. Gough, H. Klaeren:** *Executable Assertions and Separate Compilation.* In: Proceedings of Joint Modular Languages Conference, JMLC'97, Springer, 1997, pp. 41-52

[JM97] **J. M. Jézéquel, B. Meyer:** *Put it in the contract: The lessons of Ariane*. http://www.eiffel.com/doc/manuals/technology/contract/ariane/index.html, IEEE Computer, Vol. 30, No. 2, January 1997, pp. 129-130

[Kun96] **Th. Kunst:** *BTS Sidepanel Software*. Entwickler-Handbuch. sd&m GmbH & Co. KG, Munich, April, 1996

[LG86] **B. Liskov, J. Guttag:** *Abstraction and Specification in Program Development*. MIT Press, Cambridge, Mass., 1986

[Mey88] **B. Meyer:** *Object-oriented Software Construction*. Prentice Hall, 1988

[Nana] **Nana home page:** *Improved support for assertions and logging in C and C++*. http://www.cs.ntu.edu.au/homepages/pjm/nana-home/

[Sather] **Sather home page:** http://www.icsi.berkeley.edu/~sather/

[Spu94] **D. A. Spuler:** *C++ and C Debugging, Testing, and Reliability*. Prentice Hall, 1994

[TLR95] **TLR/IHK:** *Entwicklerhandbuch*. Version 4.0, sd&m GmbH&Co. KG, Munich, 1995