The Role of Contracts in Distributed Development

Martin Nordio[1], Roman Mitin[1], Bertrand Meyer[1], Carlo Ghezzi[2], Elisabetta Di Nitto[2], and Giordano Tamburrelli[2]

[1] ETH Zurich, Switzerland
{Martin.Nordio,Roman.Mitin,Bertrand.Meyer}@inf.ethz.ch
[2] Politecnico di Milano, Italy
tambug@gmail.com,carlo.ghezzi@polimi.it,dinitto@elet.polimi.it

Abstract. Distributed software development raises new software engineering challenges resulting from the difficulty of making several teams cooperate across different countries, time zones and cultures. These obstacles can lead to critical delays or even failures. One of the most effective techniques for overcoming them is to improve the quality of software specifications. Our experience with a distributed software project in an educational environment suggests that Design by Contract techniques provide a promising solution.

Key words: Software Requirements Specifications, Distributed Development, Interface Specifications, Contracts

1 Specifications in Distributed Software Development

Whether outsourced [24] or not, todays software projects are ever more often *distributed*: developed by two or more teams working in different locations. Distributed software development poses new software engineering challenges; previous work has, for example, analyzed how to adapt the old idea of "code reviews" to this new setup [25]. Here we consider another difficulty of distributed software development: how to mitigate the risk of *misunderstanding software specifications*.

The case of particular interest is the sharing of specifications between a "client" team which needs a certain functionality and a "supplier" team which implements that functionality. We will present the use of Design by Contract techniques [23, 27] to express the specifications in a precise yet understandable way, acceptable to both client and supplier teams.

Section 2 describes the source of the experiments described here: distributed software projects involving teams from different universities. Section 3 presents some of the typical problems encountered in the absence of a systematic approach to specification. Section 4 solves these problems using contracts. Section 5 describes how our approach has been applied to distributed projects. Finally, we present the results, related work, and the lessons learnt during the project.

2 Context of this Study

While some of the authors have applied the techniques described here in commercial distributed developments, the experience underlying this article is based on an academic effort rather than an industrial project.

For several years the Chair of Software Engineering at ETH Zurich has taught a course entitled *"Distributed and Outsourced Software Engineering"* or *DOSE*[1], which since 2007 has included a course project pursued in cooperation with other universities, most recently Politecnico di Milano (Italy), Odessa National Polytechnic (Ukraine), the State University of Nizhny Novgorod (Russia), and University of Debrecen (Hungary). While each university retains its own course and organization, the project is shared: each project group includes teams from different universities. Specifically, each group in the current setup is made of three teams, each including two students from a given university. (This terminology is needed to understand the rest of the discussion: a *group* does the full project and is made of teams, each doing a part of the project; a *team* is made of students from one university, but a group involves teams from different universities.) All software is developed in Eiffel using the EiffelStudio Integrated Development Environment.

As a result of this project scheme, the students get to experience the challenges of true distributed development; they face the same difficulties as in a distributed project in industry, compounded by the specific constraints of a university environment. As an example of where an "academic" setup can in fact be tougher than an industrial one, the option of delaying the final delivery (an event that, although undesirable, often happens in industry) is not available: come rain or shine, the university administration requires instructors to give the students a grade at the end of the semester, a milestone that cannot be moved.

The course allows students to experience first-hand the tasks and challenges of modern software development, and learn critical skills; they consistently report that it is a richly rewarding experience. It also provides us with an opportunity to study issues of distributed development in a controlled environment.

One of these issues is the difficulty of communicating requirements. The difficulty is well known to anyone who has practiced industrial software development; it is also intuitively clear that project distribution increases it. Our experience provides concrete evidence of this phenomenon, as will now be described.

3 Specification-related Errors in Distributed Development

An example from the 2007 session of the DOSE course [1] illustrates the specification risks of distributed development.

The project topic was the development of a system to analyze email postings of computer science events, in mailing lists such as the ECOOP list and SE

[1] Until 2006 the course was called "Software Engineering for Outsourced and Offshore Development".

World, to feed the Computer Science Event List (CSEL) [8], a Web page of Informatics Europe (http://events.informatics-europe.org). The automatic part of the system must identify key elements of a conference announcement, such as event name, event date and call for papers deadline, to prepare a CSEL entry. Since the identification cannot be perfect, the system includes a human editing step to correct any mistakes.

Figure 1 shows the Software Requirements Specification (SRS) as given to the students.

The teaching team divided the system into three clusters (subsystems):

- *A - ANALYZE:* automatically extract the essential information.
- *B - BEFIT:* user interface for interactive correction.
- *C - COMBINE:* : integration of components *A*, *B* and the CSEL website.

Correspondingly, each project group was divided into three teams, each from a given university, for example two teams performing task *A* and *B* in Zurich, and a team *C* in Odessa.

While undoubtedly not perfect, the requirements document of Figure 1 was written carefully and would appear to be clear enough. When given to teams working in different locations, however, it led to misunderstandings that the specification literature has analyzed [22]. In particular, the following problems arose. (We use the phrase "Team *A*" to mean "The team in charge of implementing cluster *A* in one of the groups," and similarly for other clusters. Different examples may involve different groups.)

Case 1. Team *A* implemented the abstract deadline using the date format *day.month.year* where *day*, *month*, and *year* are integers. Team *B* used a different format, with integers for the day and year but a string (such as "January" or "February") for the month. This misunderstanding, affecting the type of an attribute, caused a delay in the integration. It can be traced to a lack of precision of the specification (the SRS).

Case 2. Team *C* realized that the abstract submission deadline must always be earlier than the paper submission deadline. Thus, they checked this property before submitting the conference information to CSEL. If this property did not hold, an exception was triggered. Team *B*, in devising the user interface, did not check for this property and accepted any combination of dates. As a result, some combinations crashed the system. A similar problem happened to another group with the starting and the ending conference dates. The problem here is the specifications failure to state a requirement which appears necessary to someone trying to understand the system semantics.

Case 3. Team *A* understood that the category of a conference is "Conference" **or** "Symposium" **or** "Workshop" **or** "Summer School", where **or** is the usual, non-exclusive boolean disjunction. Team *B* interpreted it as an exclusive or. As a result, some test cases passed the checks performed in cluster *A* but not those of *B*, again triggering run-time exceptions and failure. The problem here is the lack of precision of natural language.

Case 4. The teams used a class called *EVENT* to model the notion of conference, but had slightly different interpretations of the semantics of this class.

A. Scope
The system shall identify the elements of a call for paper posted in mailing lists, and feed them to the CSEL system by sending e-mails in the special format.

B. Definitions, Acronyms and Abbreviations
CSEL: Computer Science Event, *http://www.informatics-europe.org/cgi-bin/informatics_events.cgi*
Conference Name: Name of the event.
Conference Dates: Starting and ending dates of the event.
Abstract Deadline: The date for the abstract submission.
Submission Deadline: The date for the paper submission.
Conference Category: Kind of the event (symposium, conference, workshop, etc).

C. Product functions
The system shall
C.1. Provide functionalities to extract the information of a conference from an e-mail (a text e-mail, no html);
C.2. Report the extracted information in a graphical user interface (GUI);
C.3. Allow modifying this information;
C.4. Submit the information to the CSEL system by sending e-mails.

D. Specific requirements
D.1. The system shall be able to extract the elements of a call for paper from text e-mails. The elements of a call for paper are the following: (1) Conference name, (2) Conference dates, (3) Abstract deadline, (4) Submission deadline, (5) Place where the conference takes place, (6) URL of the conference, (7) Conference sponsor, (8) Contact information, (9) Keywords of the conference, and (10) Conference category.
D.2. The conference category is either "*Conference*" or "*Symposium*" or "*Workshop*" or "*Summer School*".
D.3. The system shall visualize conference information, and allow modifying it. The system shall feed the approved information by sending e-mail to *CSEL* as a comma separated list.
D.4. All the elements from D.1 must be in the e-mail. If any of this information could not be extracted, the system shall add the keyword *NONE* in corresponding element.
D.5. The system can send the e-mail only if at least all key elements have been extracted or introduced by the user. The key elements are: (1) conference name, (2) conference dates, (3) abstract deadline, (4) submission deadline, (5) place where the conference takes place, and (6) URL of the conference.

Fig. 1. Example Software Requirements Specification.

In the view of Team *C*, class *EVENT* only models conferences that satisfy basic validity constraints, such as the Call for Papers deadline appearing before the notification date. Teams *A* and *B* assumed that the class models any conference, even one with invalid information; they checked the validity of the information before submitting it to CSEL. These conflicting conventions were discovered late in the project and delayed integration. The problem in this case is not in the

original requirements specification but in the lack of precision of module interface specifications produced during the design phase.

In this 2007 session of the DOSE course, no project succeeded in producing a system that could be actually deployed, although at least one came tantalizingly close; it was probably a week or two away from success but, as noted, there is no possibility of extension in a university course context. In our analysis the main reason for this result is the accumulation of specification issues such as the above, each small in itself but leading to mistakes and delays. That so many such issues could arise in a small system with a fairly straightforward specification gives an idea of the trouble insufficient specification techniques can cause in large industrial software developments.

4 Using Contracts to avoid Specification Errors

Avoiding the kind of problems illustrated above involves technical and non-technical measures. As an example of the latter, it is always desirable to check the requirements for satisfaction of the properties listed in the IEEE Standard on Requirements specification [15], such as absence of ambiguity. Such goals are, however, quite general, and the standard does not specify how to achieve them and assess the results.

Using a *formal specification* technique would remove ambiguity and help achieve some of the other quality goals. A fully formal approach is, however, beyond the reach of most teams.

Design by Contract techniques retain some of the benefits of formal methods but are far easier to teach to developers who are competent software engineers (or, in our case, software engineering or computer science students) but have not necessarily received special formal methods training.

The basic idea of Design by Contract [23, 27] is to attach partial but rigorous specifications to software elements: preconditions and postconditions for routines, and (in an object-oriented) invariants for classes. Design by Contract has applications to software construction, documentation, testing (in particular with the recent development of automatic testing tools such as AutoTest [26, 5]), proper use of programming mechanisms such as inheritance and exception handling, and management. The application of most interest here is to the specification of module interfaces.

Specifications using Design by Contract use a subset of the programming language (typically Eiffel, but others have been proposed, such as Spec# [3] and JML [18, 19]); assertions (contract) elements are boolean expressions, with some extensions such as the **old** notation in postconditions.

The class interface in Figure 2, expressed in Eiffel, describes the notion of event as managed in our example system.

```
1 indexing
    description: "Technical events as managed in the CSEL."
3
  class
5     EVENT

7 feature  -- Basic operations

9    submit_to_csel
                -- Submit the conference information by sending an e mail.
11       require
             valid_conferences: starting_date . earlier_than (ending_date)
13           valid_deadlines: abstract_deadline . earlier_than (paper_deadline)
       do
15     end

17 feature  -- Implementation

19   name: STRING
     starting_date, ending_date: DATE
21   abstract_deadline, paper_deadline: DATE
     place, url, sponsor, keywords: STRING
23   a_category: CATEGORY

25 invariant
       category_status: a_category. is_conference xor
27                      a_category. is_symposium xor
                        a_category. is_workshop  xor
29                      a_category. is_summer_school

31 end
```

Fig. 2. Interface Specification of a Class *EVENT*.

Class *EVENT* relies on an auxiliary class *CATEGORY* (presented in Figure 3).

```
indexing
    description: "Conference categories."

class
    CATEGORY

feature  -- Status report

    is_conference: BOOLEAN
            -- Does this category represent conferences?
        do
        end

    is_symposium: BOOLEAN
            -- Does this category represent symposiums?
        do
        end

    is_workshop: BOOLEAN
            -- Does this category represent workshops?
        do
        end

    is_summer_school: BOOLEAN
            -- Does this category represent summer schools?
        do
        end

end
```

Fig. 3. Interface Specification of a Class *CATEGORY*.

The actual class texts will contain implementations of the features involved (*submit_to_csel* etc.); the above are interface specifications, which can be written first and then refined into the implementations, or extracted automatically (by tools of the development environment) from these implementations if they already exist.

Class *EVENT* as given serves as a precise specification of the notion of event, avoiding the errors and ambiguities that occurred during the 2007 project development cited above. Note in particular how the class invariant expresses, through the use of the exclusive-or operator **xor**, that the different categories of event are exclusive. The precondition (**require** clause) of procedure *submit_to_csel* states validity requirements: the starting date must precede the ending date, and the deadline for abstracts must precede the deadline for papers.

5 Improving the Project Setup

The preceding example suggests that a systematic use of contracts can provide considerable help towards solving the specification and communication issues that plague distributed projects. We used the 2008 session of the DOSE course to assess this conjecture.

A number of characteristics changed between the 2007 and 2008 sessions. DOSE 2007 [1] had, as noted, the CSEL system as the project theme. The project, developed over 11 weeks out of the semesters 13, was divided into four phases:

- *Phase 1:* Write specification of each cluster (4 weeks).
- *Phase 2:* Revise and consolidate the specification into one project document; develop interface specification using contracts (3 weeks).
- *Phase 3:* Implement clusters (2 weeks).
- *Phase 4:* Test system (2 weeks).

As indicated for Phase 2, students were encouraged to use contracts, but this was only a recommendation. Faced with the difficulties mentioned earlier, students gradually realized the importance of precise specifications and started applying contracts more systematically. In the end, however, the delay in applying these techniques made it impossible to integrate the results into a deployable system.

DOSE 2008 [2] used a different project. We took advantage of the announcement of a competition in conjunction with the 2009 International Conference on Software Engineering (ICSE): the SCORE project competition [32]. Specifically, we chose one of the topics offered in the SCORE competition: "BTW" [28], a system to provide advice to someone planning a trip to a city. As in 2007, we divided the project into three clusters to be handled by different teams within a group; the *BTW* cluster were:

- *Cluster 1 - SYST:* GUI and overall organization of the *BTW* system
- *Cluster 2 - GEO:* Interface with *GIS* information and Traffic
- *Cluster 3 - PLAN:* Route planning and advice

and typical group configurations were:

- (1) Zurich - (2) Nizhny Novgorod - (3) Milano
- (1) Debrecen - (2) Milano - (3) Zurich
- (1) Milano - (2) Zurich - (3) Odessa

The problem domain made it possible to take advantage of an existing system for city modeling and route planning, the Traffic library [20], developed at ETH for the purposes of our introductory programming course [16].

While the overall setup was similar to the 2007 session, we changed a number of elements in light of the lessons learned. We started the project earlier, so that it could use 13 weeks out of the semesters 14. Recognizing the importance and difficulty of the specification phase, we extended it to 5 weeks and simplified the process by bringing the number of phases to three:

– *Phase 1:* Write specification of each cluster (4 weeks).
– *Phase 2:* Revise and consolidate the specification into one project document; develop interface specifications using contracts (5 weeks).
– *Phase 3:* Implement clusters (4 weeks).

We gave much more precise and prescriptive recommendations to students:

– They were told to get in touch with the other teams in the very first week; this avoided communication issues and simplified the revision of the requirements document.
– We introduced a code review to improve the interface specification.
– Students had to implement the projects in two cycles, which helped to find integrations problems earlier.
– We strongly encouraged them to commit the code daily, and to define and apply precise commit rules (such as permitting commit *only if the code has been compiled and tested*).

Most importantly, we made the inclusion of contract interface specifications mandatory in the specifications.

6 Results

The results of the 2008 projects confirmed the usefulness of the measures described above. The final result of the implemented projects was good: the systems were integrated and the three clusters worked in the same system. The specification of the interfaces was improved, and contracts helped to document and understand the interfaces.

To obtain the students perspective we asked them to fill a feedback questionnaire, which most of them (95%) did.

Most of the students think that contracts helped to develop the project. We wanted to know how much effort the contracts required. Table 1 shows the hour/person per team expended in developing the requirements documents with interface specification using contracts. In average, the development of contracts took 22.2% of the time used in the requirement phase.

The results of the experience show that contracts were key to develop distributed projects. The use of contracts in SRS have been useful not only to avoid misunderstandings but also to specify the interaction between subsystems. Projects that defined good interfaces using contracts have been able to deploy, and produce a final system. On the other side, projects that have not specified the interfaces properly have failed to produce a final system.

To go beyond such assessments, we intend to perform a more objective measurement of the specification effort as part of DOSE 2009.

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	**Average**
Person/hours SRS (without contracts)	35	64.6	116	108	39	82	27	34	19	89	28	22	**55.3**
Person/hours writing contracts	20	15	20	20	8	30	8	4	20	30	7	8	**15.8**
Percentage in writing contracts	36.3	18.8	14.7	15.6	17.0	26.7	22.8	10.5	51.2	25.2	20	26.6	**22.2**

Table 1. Effort expended developing requirements documents and interfaces with contracts.

7 Related Work

Industry and academia have been interested in distributed development. Lessons learnt on educational experiences have been reported [9, 7, 4]. Gotel et al. [9, 10] describe the lessons learned from the development of a project across three globally distributed educational institutions. The institutions that participated in that project were Pace University (US), University of Delhi (India), and Institute of Technology of Cambodia (Cambodia). They discuss the problems faced in the projects such as communication (with a twelve hours time difference), project planning, and cultural aspects. A similar experience is described by Damian et al. [7]. They report on the teaching experience developing software requirements specifications in geographically distributed software development with three universities (located in Canada, Australia, and Italy), focusing on the times zones and the cultural differences. These works focus on how to teach a course in distributed software development. However, they do not cover how to improve software requirements specifications. A deeper description of existing works concerning global development and educational experiences is beyond the scope of this paper and can be found in [4, 12, 13, 29].

Corriveau [6] indentifies the key properties that a contract between the parties involved in outsourcing must satisfy. These properties are expressed with a model, and this model must be testable, executable, and abstract. This model is used to test the quality of the developed project, but it does not help to understand the system under development. Our approach is used to solve the problems of potential misunderstandings in software requirements specifications, and to improve them. The use of contracts brings the same properties: testable, executable, and abstraction. Meyer et al. [21] have described our first experience in distributed software development, DOSE 2007. They described the experiences of software engineering projects in local and distributed developments. However, the role of contracts in software requirements specification is not discussed.

Sutherland et al. [33] report the industrial experience of developing a distributed project with two companies: SirsiDynix (Utah, US) and StarSoft De-

velopment Laboratories (St. Petersburg, Russia). They analyze and recommend best practices for globally distributed agile teams. They report that distributed teams can be as productive as a small collocated team.

Concerning program specifications, many existing works address this issue from different points of view and with different goals. The fact that natural language specifications lead to unsatisfactory and ambiguous specifications is well known and widely accepted. This issue becomes crucial in distributed or global development settings. Nevertheless, specifications based on natural language descriptions even if supported by diagrams (e.g., UML [34]) are still widely adopted in industrial development. Consequently, several existing approaches aim at supporting software development with specifications based on natural language [14, 31], however, the most promising techniques rely on the adoption of formalisms.

Languages such as Alloy [17] can be used to solve the problems of ambiguous specifications. However, the specification is completely detached from the source code of the program leading to a traceability gap among code and its specification. Moreover, concerning UML, it is important to notice the difference among writing contracts in Eiffel and writing constraints in the Object Constraint Language (OCL) [34]. First of all, the former approach offers a precise and non ambiguous semantics conversely to the UML object constraint language (several approaches addressed this issue, e.g. [30]). Secondly, OCL suffers from a traceability gap between the specifications and the implementation, while our approach does not.

The Java Modeling Language (JML) [18, 19] is a behavioral interface specification language that can be used to specify the behavior of Java modules. It combines the design by contract approach of Eiffel and the model-based specification approach of the Larch family of interface specification languages [11]. Although the approach is similar to Eiffel contracts, specifications in JML are not part of the Java language. Furthermore, Spec# [3] extends C# with formal specifications. In our approach, the specifications are not restricted to any programming language, thus JML and Spec# can be used to specify the interfaces of distributed projects.

8 Conclusions and DOSE 2009

We have presented an approach to improve software requirements specifications. This approach integrates contracts to SRS. To measure the results of this approach, we have developed several distributed projects. Contracts have helped to solve the problems of misunderstanding and under specification in SRS. The use of contracts brings the advantages of automatic testing and better system documentation. Although the experiments were performed in an academic environment, we believe that the results are also interesting to industrial software developers.

Since the distributed projects in DOSE 2007 and DOSE 2008 have been an interesting experience, we plan to continue this experiment in 2009. So far, we

have developed projects with two and three geographically different locations. During DOSE 2009 we have observed that projects distributed in two locations have less overhead in communication and development than projects developed in three locations. However, we have not executed any empirical study that shows what is the overhead. Next year, we plan to analyze this overhead in communication and development when projects are distributed in two, three, and four locations. If you are a member of an academic institution and would like to be part of DOSE 2009, please contact us to discuss and organize your participation.

Acknowledgements

We would like to thank all the people involved in DOSE 2007 and DOSE 2008 especially Dr. Peter Kolb, Prof. Viktor Gergel, Andrey Zaychikov, Lajos Kollr, Prof. Juhsz Istvn, and Prof. Victor Krissilov; to the students who worked hard and gave us useful feedback; to Scott West for reviewing and providing helpful comments on drafts of this paper.

References

1. DOSE 2007. http://se.ethz.ch/teaching/2007-f/outsourcing-0273/index.html.
2. DOSE 2008. http://se.ethz.ch/teaching/2008-h/dose-0273/index.html.
3. M. Barnett, K.R.M. Leino, and W. Schulte. The Spec# programming system: An overview. *Lecture Notes in Computer Science*, 3362:49–69, 2005.
4. B. Bruegge, A.H. Dutoit, R. Kobylinski, and G. Teubner. Transatlantic project courses in a university environment. In *7th Asia-Pacific Software Engineering Conference (APSEC 2000)*, pages 30–37, 2000.
5. Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. ARTOO: Adaptive Random Testing for Object-Oriented Software. In *Proceedings of the 30th International Conference on Software Engineering 2008 (ICSE'08)*, May 2008.
6. J.P. Corriveau. Testable Requirements for Offshore Outsourcing. In *Software Engineering Approaches for Offshore and Outsourced Development (SEAFOOD)*, volume 4716, pages 27–43. Springer, 2007.
7. D. Damian, F. Lanubile, and T. Mallardo. Investigating IBIS in a Distributed Educational Environment: the Design of a Case Study. In *Workshop on Distributed Software Engineering*, volume 1, 2005.
8. Computer Science Event:. http://www.informatics-europe.org/cgi-bin/informatics_events.cgi.
9. O. Gotel, V. Kulkarni, L.C. Neak, C. Scharff, and S. Seng. Introducing Global Supply Chains into Software Engineering Education. In *Software Engineering Approaches for Offshore and Outsourced Development (SEAFOOD)*, volume 4716, pages 44–58. Springer, 2007.
10. Olly Gotel, Vidya Kulkarni, Christelle Scharff, and Longchrea Neak. Students as Partners and Students as Mentors: An Educational Model for Quality Assurance in Global Software Development. In K. Berkling, M. Joseph, B. Meyer, and M. Nordio, editors, *Software Engineering Approaches for Offshore and Outsourced*

Development (SEAFOOD 2008), volume 16 of *Lecture Notes in Business and Information Processing*. Springer-Verlag, 2009.
11. JV Guttag, JJ Horning, SJ Garl, KD Jones, A. Modet, and JM Wing. Larch: languages and tools for formal specification. *Texts and Monographs in Computer Science*, 1993.
12. M.J. Hawthorne and D.E. Perry. Software engineering education in the era of outsourcing, distributed development, and open source software: challenges and opportunities. In *International Conference on Software Engineering*, volume 27, page 643. Springer, 2005.
13. J.D. Herbsleb and D. Moitra. Global software development. *Software, IEEE*, 18(2):16–20, Mar/Apr 2001.
14. A. Holt. Formal verification with natural language specifications: guidelines, experiments and lessons so far. *South African Computer Journal*, pages 253–257, 1999.
15. IEEE:. IEEE Recommended Practice for Software Requirements Specifiations. IEEE Std 830, 1998.
16. Introduction to Programming (Einführung in die Programmierung) - Chair of Software Engineering - ETH Zurich. http://se.ethz.ch/teaching/2008-h/eprog-0001/index.html.
17. D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.
18. G. Leavens, A. Baker, and C. Ruby. JML: A notation for detailed design. *Kluwer International Series in Engineering and Computer Science*, pages 175–188, 1999.
19. G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.
20. Traffic Library:. http://traffic.origo.ethz.ch/.
21. B. Meyer and M. Piccioni. The allure and risks of a deployable software engineering project. In *Proceedings of the 21st IEEE-CS Conference on Software Engineering Education and Training*, 2008.
22. Bertrand Meyer. On formalism in specifications. *IEEE Software*, 2(1):6–26, 1985.
23. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.
24. Bertrand Meyer. The unspoken revolution in software engineering. *IEEE Computer*, 39(1):121–124, 2006.
25. Bertrand Meyer. Design and Code Reviews in the Age of the Internet. In K. Berkling, M. Joseph, B. Meyer, and M. Nordio, editors, *Software Engineering Approaches for Offshore and Outsourced Development (SEAFOOD 2008)*, volume 16 of *Lecture Notes in Business and Information Processing*. Springer-Verlag, 2009.
26. Bertrand Meyer, Ilinca Ciupa, Andreas Leitner, and Lisa (Ling) Liu. Automatic testing of object-oriented software. In Jan van Leeuwen, editor, *Proceedings of SOFSEM 2007 (Current Trends in Theory and Practice of Computer Science)*, Lecture Notes in Computer Science. Springer-Verlag, 2007.
27. Bertrand Meyer (editor). ISO/ECMA Eiffel standard (Standard ECMA-367: Eiffel: Analysis, Design and Programming Language), June 2006. available at http://www.ecma-international.org/publications/standards/Ecma-367.htm.
28. BTW Project:. http://score.elet.polimi.it/projects.html.
29. Ita Richardson, Allen E. Milewski, Neel Mullick, and Patrick Keil. Distributed development: an education perspective on the global studio project. In *ICSE '06:*

Proceedings of the 28th international conference on Software engineering, pages 679–684, New York, NY, USA, 2006. ACM.
30. M. Richters and M. Gogolla. On formalizing the UML object constraint language OCL. *Lecture Notes in Computer Science*, pages 449–464, 1998.
31. M. Saeki, H. Horai, and H. Enomoto. Software development process from natural language specification. In *ICSE '89: Proceedings of the 11th international conference on Software engineering*, 1989.
32. SCORE:. http://score.elet.polimi.it/.
33. J. Sutherland, A. Viktorov, J. Blount, and N. Puntikov. Distributed scrum: Agile project management with outsourced development teams. In *HICSS'40, Hawaii International Conference on Software Systems*, 2007.
34. UML. http://www.uml.org/.