Software Certification - Coding, Code, and Coders

Klaus Havelund and Gerard J. Holzmann
Laboratory for Reliable Software (LaRS)

Jet Propulsion Laboratory, California Institute of Technology
4800 Oak Grove Drive, Pasadena, California, 91109-8099
firstname.lastname@jpl.nasa.gov

ABSTRACT

We describe a certification approach for software development that has been adopted at our organization. JPL develops robotic spacecraft for the exploration of the solar system. The flight software that controls these spacecraft is considered to be *mission* critical. We argue that the goal of a software certification process cannot be the development of "perfect" software, i.e., software that can be formally proven to be correct under all imaginable and unimaginable circumstances. More realistically, the goal is to guarantee a software development process that is conducted by knowledgeable engineers, who follow generally accepted procedures to control known risks, while meeting agreed upon standards of workmanship. We target three specific issues that must be addressed in such a certification procedure: the coding process, the code that is developed, and the skills of the coders. The coding process is driven by standards. The code is mechanically checked against the standards with the help of stateof-the-art static source code analyzers. The coders, finally, are certified in on-site training courses that include formal exams.

Categories and Subject Descriptors

D.2.0 [Software Engineering]: General – standards. K.5.2. [Governmental Issues]: – Regulation. K.7.3: [The Computing Profession]: Certification, Licensing and Testing.

General Terms

Design, Reliability, Standardization, Verification, Legal Aspects.

Keywords

Coding standards, code review, static source code analysis, logic model checking, unit testing, safety- and mission-critical software.

1. INTRODUCTION

John Rushby once described the dilemma of current approaches to software verification or certification as follows: "Because we cannot demonstrate how well we've done, we'll show how hard we've tried." [1]. The statement is apt. Few, if any, organizations feel confident enough about their software development processes that they are willing to give an absolute guarantee of its fitness for use or so much as the absence of preventable flaws in workmanship.

As customers, and generally as users that have to rely on the safety and reliability of sometimes critically important software applications (e.g., as used by banks, car makers, or in medical

Copyright 2011 Association for Computing Machinery. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the U.S. Government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

EMSOFT'11, October 9–14, 2011, Taipei, Taiwan. Copyright 2011 ACM 978-1-4503-0714-7/11/10...\$10.00. devices), we are quite used to the opposite: we are routinely asked to sign disclaimers that hold the software makers invulnerable to flaws in workmanship and all possible damage that might be caused by it. This in itself is remarkable.

At some point, perhaps a few decades ago, we may have expected that standard market principles would solve this problem: customers could have been expected to reject products that are delivered without warranties of fitness. But this is not what happened.

The driving principles that determine how software applications are developed and marketed give a significant advantage to the vendor who delivers a new service first, and merely commits to slowly improve while the product is in use, based on customer feedback. The customers, in this way, become part of what otherwise would be the testers, except this group of testers pays the vendor, instead of the reverse. As unsettling as this might be from a philosophical point of view, it works quite well for the vast majority of software products sold today.

A clear exception holds for the category of *safety-critical* and *mission-critical* software applications (the distinction is whether a system failure may result in death/serious injury to people versus loss of mission). Most will agree that different rules must apply here, since for these types of applications it cannot be considered acceptable for a vendor to decline all responsibility for the potential damage caused in return for a mere commitment to fix any problems not caught in the software development process until *after* they have manifested themselves to end-users. If we now look more carefully at *which* different rules are applied in these cases, we are in for a surprise. In many cases there are no software certification requirements, and those requirements that do exist can only be described as modest. Organizations are often only asked to show "how hard they've tried" and not that certain standards of workmanship are met.

As part of our research and work, we have inevitably gained experience with the analysis of many software products that are considered critical. At JPL this naturally includes the analysis of the control software for interplanetary spacecraft, but we have also been involved in a broad range of other types of safety-critical applications, including the investigation of specific aspects of automotive software (e.g., in the context of a study of the potential for sudden unintended acceleration of Toyota vehicles in 2010), medical device software, software used in the shutdown systems of nuclear power plants, railway signaling protocol software, etc.

NASA's shuttle software [2] is often mentioned as an example of how critical software systems can reach a high level of safety and reliability. This software indeed has an exemplary track record of having a low residual fault density rate over the approximately three decades of use. Like any other human design, it is, of course,



not completely free from defects, nor can it be expected to be. One could well say that the first principle adopted in the design of any system that is meant to be reliable is the recognition that no single system component can be perfect: every part has breaking points, some known and some unknown. Reliability and safety, therefore should be treated as system properties, not component properties. To complete the argument, in almost all cases of interest the software is merely one component in a larger system that includes also hardware and human operators as essential elements.

The software used for commercial airplanes, similarly, has an enviable track record for reliability. Again, the track record is not for perfection, because in any sufficiently complex system there are always residual defects that are discovered only after a system is delivered and goes into operation. The goal for certified software, therefore, cannot be to put a process in place that guarantees correctness under all circumstances - the goal is to produce a safe and reliable system that is build by competent, well-trained developers, following a process that controls risks and meets the evolving standard of skilled workmanship. In one sentence here, then we touch on three separate targets for a software certification process: the coding process that is followed, the code, and the coders. The certification process that is followed in the aerospace industry (e.g., for software used to control commercial airplanes), targets primarily the coding process, e.g. with standards such as DO-178B [3]. There are no strict requirements here for the use of specific verification tools, or for the certification of software developers themselves. The target is only to secure that due diligence was used in the development process itself.

2. CERTIFICATION PROCESS

At JPL, in the development of the control software for interplanetary spacecraft, such as the Mars Exploration Rovers [4], we have adopted a different process. The intent of this process is to subject not just the coding process, but also the code, and the coders, and to some extent the software managers as well, to some form of certification.

2.1 The Coding Process

For flight code, JPL has adopted an Institutional Coding Standard [5], with which it requires compliance in all newly developed mission-critical software written at JPL. Most flight software, by a significant margin is traditionally written in the C programming language, and therefore the JPL coding standard for flight software targets this language. The coding standard deliberately captures only risk-related rules for which compliance can be verified mechanically. Other than most other coding standards, then, this standard has real teeth: except in rare cases, noncompliance is not an option for our flight software developers. Because all rules in the standard are specifically risk related (i.e., we can often point at a mission anomaly or mission loss that was caused by the violation of the underlying principle), approval for non-compliance is also rarely requested or granted. An example of a risk-related rule in this coding standard is the abolition of all dynamic memory use and of recursive code. Some of the motivation for the rules can be traced to the Power of Ten rules, described in [6]. JPL further imposes fairly strict requirements on the code review process some of which is detailed in [7]. The coding standard for C is included in brief format in Appendix A, organized according to importance of the rules.

JPL has also developed a coding standard for Java [8]. Java is at JPL mostly used for ground software. That is, software that executes on ground-based computers in mission operation centers, for example receiving telemetry from and sending commands to the rovers. Although this is not embedded software, its correct behavior is important for the correct behavior of the rovers. The Java coding standard is in a draft form and is not yet an institutional standard. It is more liberal in certain areas than the C coding standard. For example, it does allow dynamic memory allocation (use of the **new** construct). The coding standard for Java is included in brief format in Appendix B, organized according to subject categories. The rules represent headlines, which in the full standard are explained in more detail.

2.2 The Code

The code is rightfully subject to the strictest requirements. Flight code, e.g. for the MSL mission [4], is checked nightly for compliance with the JPL C coding standard [5], and subjected to rigorous analysis with four separate state-of-the-art static source code analysis tools [7] (at the time of writing this includes the commercial tools [9]: coverity, codesonar, and odasa (from semmle), and the research tool uno). The warnings generated by each of these tools is combined with the output of missionspecific checkers that secure compliance with naming conventions, coding style, etc. In addition, all warnings, if any (there should be none), from the standard C compiler, used in pedantic mode with all warnings enabled, are included in the results that are provided to the software developers as part of the standard 'scrub' interface [7]. The developers are required to close out all reports before a formal code review is initiated. In peer code reviews, an additional source of input is provided by designated peer code reviewers, and added to the 'scrub' results.

Furthermore, each programmer is responsible for writing unit tests for his/her modules. A compilation build of the entire system includes running all unit tests, which have to succeed for the build to succeed. Separately, key parts of the software design are also checked for correctness and compliance with higher level design requirements with the help of logic model checkers, such as Spin [10]. Training in the use of logic model checkers is tacitly provided via (optional) graduate-level courses taught by members of the JPL Laboratory for Reliable Software in the Computer Science Department at the California Institute of Technology. Approximately ten JPL employees outside the Laboratory for Reliable Software have so far taken and passed this course, and have become proficient in the use of logic model checkers for the analysis and verification of flight software.

The Java coding standard in its entirety has in collaboration with semmle been implemented in semmle's static code analyzer, and is currently being tested on JPL internal projects, including the MSL telemetry and command ground software. The implementation is being refined (in collaboration with semmle), driven by the results obtained during these tests. The refinement consists of finding the right balance between a low number of false positives and a high number of true negatives. Too many false positives will discourage use of the standard.

2.3 The Coders

Starting in 2010, JPL adopted a new procedure for the certification of flight software developers. The procedure itself is still subject to some revision, but once fully operational no software developer will be able to touch flight code (develop, manage, or modify) without having successfully completed a JPL specific Flight Software Certification course. The course consists



of three modules, focusing on (a) computing science principles, (b) JPL software development standard processes, and (c) software risk and software vulnerabilities. Each module takes two full days of instruction, for a total of six days for all three modules combined. Each module ends with an exam that must be completed with a passing grade. At the time of writing, the first twenty software developers have successfully completed this course, and have received their certificates. Others have not passed and will have to take the course, and the exams, again. New classes are held several times a year, until all software developers have been certified. At that point, we will likely add refresher courses for those who are already certified, in addition to the basic certification course itself, to keep pace with continuing developments in this field. The certification course intends to certify that all developers of critical code are familiar with basic computing science theory, and standard algorithms, are intimately familiar with the risks inherent in the use of the programming languages that are typically used for flight code, and understand not just the letter but also the rationale for the coding standard that they are expected to follow. The certification course also introduces developers to the tools (e.g., static analyzers, code review tools, and unit test tools) that they will be using in flight software development.

Perhaps as an aside, JPL has also instituted an (as yet non-required) course for senior management. Senior management normally has deep experience with spacecraft and mission design, but less so with software design principles. To date, most of JPL's senior management has taken and completed this course. The course is repeated once a year, and is by invitation only.

3. REGULATORY PROCESS

As noted in the introduction, members of our team have been involved in a broad range of software analysis applications, targeting not only aerospace but also safety-critical software used in automobiles, medical devices, and in the shutdown systems of nuclear power plants. It is perhaps noteworthy that at present there do not appear to be any strict regulatory requirements on the development of these critical types of software applications, neither on the code or the coders, on the organization that employs the coders, or on the processes that are followed in the coding process.

In the automotive industry there is reasonable consensus on at least one set of coding guidelines: the one developed by the organization MiraLtd, and known as the MISRA-C Coding Guidelines [11]. Curiously, although many developing organizations have publically expressed support for these guidelines, there is no requirement (or verification) that they actually comply with them.

Compliance with any reasonable standard, e.g., [5,6,11,8], can make it significantly simpler to analyze code for potential anomalies, and to revise, and maintain it longer term. Much the same is true in the medical device industry, where the FDA does not require compliance with any specific coding standard or software development process, and goes no further than to

recommend the use of state-of-the-art static source code analyzers as part of software development process, without actually requiring evidence that this is done. Similarly, the Nuclear Regulatory Commission has issued no comparable requirements for any software used in the shutdown systems of future nuclear power plants, nor does it seem to have plans to do so, as a key part of the licensing process.

We believe that in each of these cases the lack of requirements on software development is an omission that should be corrected. Where not following generally accepted principles for safe software development could be regarded as a lack of workmanship on the part of the developer or developing organization, with the potential effect of contributing to preventable software failure, inadequate regulation for safety-critical software systems that we all rely on could well be regarded as a failure of the regulatory process.

4. ACKNOWLEDGMENT

The research described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

5. REFERENCES

- J. Rushby. Verified Software Systems the Certification Perspective, http://www.csl.sri.com/users/rushby/slides/vsr-roadmap-cert-apr06.pdf.
- [2] C. Fishman. They Write the Right Stuff, http://www.fastcompany.com/magazine/06/writestuff.html.
- [3] RTCA-DO-178B, Software Considerations in Airborne Systems and Equipment Certification, December 1992.
- [4] Mars Science Laboratory mission (MSL), http://www.nasa.gov/mission_pages/msl/index.html.
- [5] JPL Institutional Coding Standard for the C Programming Language, http://lars-lab.jpl.nasa.gov/JPL_Coding_Standard_C.pdf.
- [6] The "Power of Ten" Coding Rules, http://spinroot.com/p10/.
- [7] G.J. Holzmann. Scrub: a Tool for Code Reviews, Innovations in Systems and Software Engineering, Vol. 6, No. 4, 2010, pp. 311-318.
- [8] JPL Draft Coding Standard for the Java Programming Language, http://lars-lab.jpl.nasa.gov/JPL_Coding_Standard_Java.pdf.
- [9] Static Source Code Analysis Tools, http://spinroot.com/static/.
- [10] G.J. Holzmann. The Spin Model Checker Primer and Reference Manual, Addison-Wesley, 2004.
- [11] MISRA-C:2004, Guidelines for the Use of the C Language in Critical Systems, Mira Ltd.



APPENDICES

A – JPL Coding Standard for C

	Coding Standard for C		
1 Language Compliance			
1	Do not stray outside the language definition.		
2	Compile with all warnings enabled; use static source code analyzers.		
2 Predictable Execution			
3	Use verifiable loop bounds for all loops meant to be terminating.		
4	Do not use direct or indirect recursion.		
5	Do not use dynamic memory allocation after task initialization.		
*6	Use IPC messages for task communication.		
7	Do not use task delays for task synchronization.		
*8	Explicitly transfer write-permission (ownership) for shared data objects.		
9	Place restrictions on the use of semaphores and locks.		
10	Use memory protection, safety margins, barrier patterns.		
11	Do not use goto, setjmp or longjmp.		
12	Do not use selective value assignments to elements of an enum list.		
3 Defensive Coding			
13	Declare data objects at smallest possible level of scope.		
14	Check the return value of non-void functions, or explicitly cast to (void).		
15	Check the validity of values passed to functions.		
16	Use static and dynamic assertions as sanity checks.		
*17	Use U32, I16, etc instead of predefined C data types such as int, short, etc.		
18	Make the order of evaluation in compound expressions explicit.		
19	Do not use expressions with side effects.		
4 Code Clari	4 Code Clarity		
20	Make only very limited use of the C pre-processor.		
21	Do not define macros within a function or a block.		
22	Do not undefine or redefine macros.		
23	Place #else, #elif, and #endif in the same file as the matching #if or #ifdef.		
*24	Place no more than one statement or declaration per line of text.		
*25	Use short functions with a limited number of parameters.		
*26	Use no more than two levels of indirection per declaration.		
*27	Use no more than two levels of dereferencing per object reference.		
*28	Do not hide dereference operations inside macros or typedefs.		
*29	Do not use non-constant function pointers.		
30	Do not cast function pointers into other types.		
31	Do not place code or declarations before an #include directive.		
5 - MISRA shall compliance			
73 rules	All MISRA shall rules not already covered at Levels 1-4.		
6 - MISRA should compliance			
*16 rules	All MISRA should rules not already covered at Levels 1-4.		

All rules are 'shall' rules (must be followed), except those marked with '*' which are 'should' rules (justified deviations allowed).



B - JPL Coding Standard for Java

1 - Process 1 Compile with checks turned on 2 Apply static analyses. 3 Document public elements. 4 Write unit tests. 2 - Names 5 De and everrife field or class names. 6 De and everrife field or class names. 7 Make imports explicit 8 De no or have excite package dependencies. 9 Obey the contract for equals(). 10 Define both equals() and hashCode(). 11 Define equals with parameter type Object. 12 Define equals with parameter type Object. 13 De not implement the Choneable interface. 14 De not implement the Choneable interface. 15 De not contract the Choneable interface. 16 De not implement over nheritance. 17 Make fields private. 18 De not as used in mutuals fields first intuitive fields from the contract of the contract o	B – JPL Coding Standard for Java		
2	1 - Process		
3 Document public elements 4 Write unit tests. 4 Write unit tests. 5 Use the standard naming conventions 6 Do not override field or class names. 3-Packages, Classes and Interfaces 7 Make imports explicit. 8 Do not have explicit. 10 Define coult equals by and hashCode(). 11 Define cquals when adding fields. 12 Define equals with parameter type Object. 13 Do not use implizers. 14 Do not misperent the Cloneable interface. 15 Do not call nonfinal methods in constructors. 16 De not call nonfinal methods in constructors. 17 Make fields private. 18 Do not use static murable fields. 19 De not use static murable fields. 19 De not may be constructors. 19 Do not use static murable fields. 10 Define cquals with reliable fields final. 10 Define may be constructors. 10 Define cquals with reliable fields final. 10 Define may be constructors. 10 Define cquals with reliable fields final. 10 Define may be constructors. 10 Define cquals with reliable fields final. 11 Define constructors. 12 Use annotations. 12 Use annotations. 12 Use annotations. 12 Use annotations. 13 Restrict method overloading. 14 Do not explain all arrays or collections. 15 Do not return multi-arrays or collections. 16 Declarations and Statements. 17 Have one concept per line. 18 De not act with a static methods. 19 De not act with a static methods. 10 Declarations and Statements. 10 Declarations and Statements. 11 Have one concept per line. 12 Be braces in control structures. 13 De not use reference equality. 14 Declarations and Statements with default. 15 Declarations and Statements with default. 16 Declarations and Statements with default. 17 Have one concept per line. 18 Declarations and Statements with default. 19 Declarations and Statements with default. 19 Declarations and Statements with default. 20 Declarations and Statements with default. 21 Declarations and Statements with default. 22 Use annotations are call values. 23 Declarations and explain point equality. 24 Declarations are reference equality. 25 Declarations are reference equality. 26 Declar			
4 Write unit tests. 5 Use the standard naming conventions. 6 D no not override field or class names. 3 - Packages, Classes and Interfaces. 7 Make imports explicit. 8 D no table very the package dependencies. 9 Obey the contract for equals(). 10 Define both equals yand hashCode(). 11 Define cquals with parameter type Object. 13 Do not use finalizers. 14 Do not use finalizers. 14 Do not use finalizers. 15 Do not call monital methods in constructors. 16 Select composition over inheritance. 4 - Fiolds 17 Make fields private. 18 Do not used in main and the fields. 19 Declare immutable fields final. 10 Initialize fields before use. 5 - Mothods 21 Use assertions. 22 Use annotations. 23 Restrict method overloading. 24 Do not suits parameters. 25 Do not return unit arrays or collections. 26 Do not call System exit. 27 Have one concept per line. 28 Use braces in control structures. 30 Do not use method structures. 31 Bertos and Statements. 32 Restrict side effects in expressions. 33 Restrict side effects in expressions. 34 Use braces in counts structures. 35 Do not ettern unit arrays or collections. 36 Les braces in counts structures. 37 Have one concept per line. 38 Do not structure and statements. 39 Do not extern unit of structures. 30 Use only solved statements with default. 31 Use braces in counts structures. 32 Do not structure and structures. 33 Restrict side effects in expressions. 34 Les constructions for non-trivial literals. 35 Make operator precedence explicit. 36 Do not use fleating point quality. 47 Use on result type in conditional expressions. 48 Perspections 49 Do not use string conscienciation operator in loops. 49 Do not use string conscienciation operator in loops. 49 Do not not see shortly units. 40 Use one result type in conditional expressions. 41 Use one result type in conditional expressions. 42 Do not ever units when a final block. 43 Do not a term the scheduler for synchronization. 44 Use generies.	2		
5	3	Document public elements.	
5 Use the standard naming conventions 6 Do not convertide field or class names 3 - Packages, Classes and Interfaces 7 Make imports explicit. 8 Do not have explicit package dependencies. 9 Obey the contract for equals (). 10 Define both equals () and hash Code (). 11 Define equals when adding fields. 12 Define equals with parameter type Object. 13 Do not use final methods in constructors. 14 Do not use final methods in constructors. 15 Do not call on mildrers. 16 Select composition over inheritance. 4 - Fields 17 Make fields private. 18 Do not use static mutable fields final. 19 Declare immutable fields final. 20 Initialize fields before use. 21 Use assertions. 22 Use amountains. 23 Restrict method overloading. 24 Do not assign to parameters. 25 Do not return unit arrays or collections. 26 Do not call System exit. 27 Illus one concept per line. 28 Use hances in control structures. 30 Use only before use. 31 End switch statements with default. 32 Use the final structures. 33 Use braces in control structures. 34 Restrict method overloading. 35 Do not return unit arrays or collections. 36 Do not return unit arrays or collections. 37 Illus one concept per line. 38 Do not such sistements with default. 39 Do not such sistements with default. 30 Use braces in countri structures. 30 Use braces in countri structures. 31 End switch statements with default. 32 Use hances in countri structures. 33 Use only spot the sistements with default. 34 Do not such statements with default. 35 Make operator procedence explicit. 36 Make operator procedence explicit. 37 Use only spot type in conditional expressions. 38 Use one result in values. 39 Do not see floating point equality. 40 Use one result period control in loops. 41 Do not convert price conversions. 42 Use one result is values. 43 Do not deep exceptions. 44 Use generics. 45 Use interfaces as types when available. 46 Use interfaces in types when available. 47 Do not greate in the scheduler for synchronization. 49 Program against dard races. 51 Do not rely on the scheduler for	4	Write unit tests.	
3 - Packagos, Classes and Interfaces	2 - Names		
3 - Packages, Classes and Interfaces 7 - R Make imports explicit. 8	5	Use the standard naming conventions.	
7 Make imports explicit 8 Do not have explete package dependencies 9 Obey the contract for equals() 10 De fine both equals() and hashCode() 11 Define squals when adding fields 12 Define causals with parameter type Object 13 Do not use finalizers 14 Do not implement the Cloneable interface. 15 Do not call nonfinal methods in constructors. 16 Select composition over inheritance. 17 Make fields private. 18 Do not use static mutable fields. 19 Declare immutable fields final. 19 Declare immutable fields final. 10 Initialize fields before use. 20 Initialize fields before use. 21 Use annotations. 22 Use annotations. 23 Restrict method overloading. 24 Do not assign to parameters. 25 Do not return all arrays or collections. 26 Do not call System exit. 27 Ob not need the final final final fields. 28 Use braces in control structures. 29 Do not have empty blocks. 30 Use braces in switch statements with default. 31 End switch statements with default. 32 Terminate fiels-for with desc. 33 Restrict side effects in expressions. 34 Use named constants for non-trivial literals. 35 Make operator precedence explicit. 36 Do not use critical point point equality. 37 Use only short-circuit ligic joperators. 38 Do not use entire procedence explicit. 39 Do not use confirm point equality. 30 Use named constants for non-trivial literals. 31 Do not use reference equality. 32 Terminate fiels-for with else. 33 Do not use reference equality. 34 Use generics. 35 Do not our tous string conactantion operator in loops. 36 Exceptions. 37 Use privative use string conactantion operator in loops. 38 Do not use reference equality. 39 Do not use reference equality. 40 Use generics. 41 Do not remove literals from collections. 41 Use generics. 42 Use generics. 43 Do not not we obtain point equality. 44 Use generics. 45 Do not remove literals from collections. 46 Use privative recoversions. 47 Use only short-circuit logic operators. 48 Do not remove literals from collections. 49 Porgram against deaf cocks.	6	Do not override field or class names.	
7 Make imports explicit 8 Do not have explete package dependencies 9 Obey the contract for equals() 10 De fine both equals() and hashCode() 11 Define squals when adding fields 12 Define causals with parameter type Object 13 Do not use finalizers 14 Do not implement the Cloneable interface. 15 Do not call nonfinal methods in constructors. 16 Select composition over inheritance. 17 Make fields private. 18 Do not use static mutable fields. 19 Declare immutable fields final. 19 Declare immutable fields final. 10 Initialize fields before use. 20 Initialize fields before use. 21 Use annotations. 22 Use annotations. 23 Restrict method overloading. 24 Do not assign to parameters. 25 Do not return all arrays or collections. 26 Do not call System exit. 27 Ob not need the final final final fields. 28 Use braces in control structures. 29 Do not have empty blocks. 30 Use braces in switch statements with default. 31 End switch statements with default. 32 Terminate fiels-for with desc. 33 Restrict side effects in expressions. 34 Use named constants for non-trivial literals. 35 Make operator precedence explicit. 36 Do not use critical point point equality. 37 Use only short-circuit ligic joperators. 38 Do not use entire procedence explicit. 39 Do not use confirm point equality. 30 Use named constants for non-trivial literals. 31 Do not use reference equality. 32 Terminate fiels-for with else. 33 Do not use reference equality. 34 Use generics. 35 Do not our tous string conactantion operator in loops. 36 Exceptions. 37 Use privative use string conactantion operator in loops. 38 Do not use reference equality. 39 Do not use reference equality. 40 Use generics. 41 Do not remove literals from collections. 41 Use generics. 42 Use generics. 43 Do not not we obtain point equality. 44 Use generics. 45 Do not remove literals from collections. 46 Use privative recoversions. 47 Use only short-circuit logic operators. 48 Do not remove literals from collections. 49 Porgram against deaf cocks.	3 - Packages, 0	Classes and Interfaces	
8	7		
9 Obey the contract for equals(). 10 Define both equals() and hashCode(). 11 Define equals when adding fields. 12 Define equals when adding fields. 13 De not use finalizers. 14 De not implement the Cloneable interface. 15 De not call nonfinal methods in constructors. 16 Select composition over inheritance. 17 Make fields private. 18 De not use static mutable fields. 19 Declare immutable fields final. 19 Declare immutable fields final. 10 Initialize fields before use. 2 Initialize fields before use. 2 Use annotations. 2 Use assertions. 2 Use assertions. 2 De not assign to parameters. 2 Do not call system exit. 3 Restrict method overloading. 4 Do not call System exit. 4 Colectarations and Statements 5 Do not cell route finalizers. 2 Use when the finalizers of	8		
11 Define equals when addring fields	9		
11 Define equals when addring fields	10		
12 Define equals with parameter type Object. 13 De not use finalizers. 14 De not implement the Cloneable interface. 15 De not call nonfinal methods in constructors. 16 Select composition over inheritance. 4 - Fields 17 Make fields private. 18 De not use static mutable fields. 19 Declare immutable fields final. 20 Initialize fields before use. 5 - Methods 21 Use assertions. 22 Use annotations. 23 Restrict method overloading. 24 De not return null arrays or collections. 25 De not return null arrays or collections. 26 De not return null arrays or collections. 27 De not acid system. 28 Use braces in control structures. 29 De not set with statements. 29 Use haves in control structures. 29 De not have may be default. 20 De not acid system. 21 End switch statements with default. 27 Taylor on the statements with default. 30 Use breaks in switch statements. 31 End switch statements with default. 32 Terminat Felse-off with else. 33 Restrict side effects in expressions. 34 Use named constants for non-frivial literals. 35 Make operator precedence equality. 36 De not use cell values. 39 De not use reference equality. 30 Use only short-circuit logic operators. 31 Use only short-circuit logic operators. 32 De not often gonit equality. 35 Use only short-circuit logic operators. 36 De not use reference equality. 37 Use only short-circuit logic operators. 38 De not use conditional expressions. 41 De not use reference equality. 42 Use on the other conditional expressions. 43 Use interfaces at types when available. 44 Use generics. 45 Use interfaces at types when available. 46 Use primitive types. 47 De not convertise from collections. 48 Use interfaces on types when available. 49 Program against datal cokes. 50 Program against datal cokes. 51 De not on the other conversions. 52 Wait and notify safely.	11		
13 Do not use finalizers.	12		
15 Do not call nonfinal methods in constructors. 16 Select composition over inheritance. 4 - Fields 17 Make fields private. 18 Do not use static mutable fields. 19 Declare immutable fields final. 19 Declare immutable fields final. 20 Initialize fields before use. 5 - Mothods 21 Use assertions. 22 Use annotations. 23 Restrict method overloading. 24 Do not assign to parameters. 25 Do not return null arrays or collections. 26 Do not call System.exit. 6 - Declarations and Statements 27 Have one concept per line. 28 Use braces in control structures. 29 Do not the summer of the structures. 29 Do not the summer of the summer	13		
15 Do not call nonfinal methods in constructors. 16 Select composition over inheritance. 4 - Fields 17 Make fields private. 18 Do not use static mutable fields. 19 Declare immutable fields final. 19 Declare immutable fields final. 20 Initialize fields before use. 5 - Mothods 21 Use assertions. 22 Use annotations. 23 Restrict method overloading. 24 Do not assign to parameters. 25 Do not return null arrays or collections. 26 Do not call System.exit. 6 - Declarations and Statements 27 Have one concept per line. 28 Use braces in control structures. 29 Do not the summer of the structures. 29 Do not the summer of the summer			
15 Select composition over inheritance.	15		
17			
17			
18		Make fields private	
19			
S - Methods			
S - Methods			
21 Use assertions		mittanze netus before use.	
22 Use annotations. 23 Restrict method overloading. 24 Do not assign to parameters. 25 Do not return null arrays or collections. 26 Do not call System exit. 6 - Declarations and Statements. 27 Have one concept per line. 28 Use braces in control structures. 29 Do not have empty blocks. 30 Use breaks in switch statements. 31 End switch statements with default. 32 Terminate if-else-if with else. 7 - Expressions 33 Restrict side effects in expressions. 34 Use named constants for non-trivial literals. 35 Make operator precedence explicit. 36 Do not use reference equality. 37 Use only short-circuit logic operators. 38 Do not use notating point equality. 40 Use one result type in conditional expressions. 41 Do not use string concatenation operator in loops. 8 - Exceptions 42 Do not drop exceptions. 43 Do not array the inconditional expressions. 44 Use generics. 45 Use generics. 46 Use primary expressions. 47 Do not remove literals from collections. 48 Restrict numeric conversions. 49 Program against deallocks. 50 Program against data races. 50 Program against deallocks. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely.		Use assertions	
23 Restrict method overloading. 24 Do not assign to parameters. 25 Do not return null arrays or collections. 26 Do not call System exit. 6 - Declarations and Statements 27 Have one concept per line. 28 Use braces in control structures. 29 Do not have empty blocks. 30 Use breaks in switch statements. 31 End switch statements with default. 32 Terminate if-else-if with else. 7 - Expressions 33 Restrict side effects in expressions. 34 Use named constants for non-trivial literals. 35 Make operator precedence explicit. 36 Do not use reference equality. 37 Use only short-circuit logic operators. 38 Do not use cotal values. 39 Do not use floating point equality. 40 Use one result type in conditional expressions. 41 Do not use string concatenation operator in loops. 8 - Exceptions 42 Do not drop exceptions. 43 Do not abruptly exit a finally block. 9 - Types 44 Use generics. 45 Use interfaces as types when available. 46 Use primitive types. 47 Do not remove literals from collections. 48 Restrict numeric conversions. 49 Program against data races. 50 Program against dator for synchronization. 52 Wait and notify safely.			
24 Do not resum null arrays or collections. 25 Do not relum null arrays or collections. 26 Do not call System.exit. 6 - Declarations and Statements 27 Have one concept per line. 28 Use braces in control structures. 29 Do not have empty blocks. 30 Use bracks in switch statements. 31 End switch statements with default. 32 Terminate if-else-if with else. 7 - Expressions 33 Restrict side effects in expressions. 34 Use named constants for non-trivial literals. 35 Make operator precedence explicit. 36 Do not use reference equality. 37 Use only short-circuit logic operators. 39 Do not use refore equality. 40 Use one result type in conditional expressions. 41 Do not use string concatenation operator in loops. 8 - Exceptions 42 Do not drop exceptions. 43 Do not drop exceptions. 44 Use generies. 45 Use generies. 46 Use primitive types. 47 Do not remove literals from collections. 48 Restrict numeric conversions. 49 Program against data races. 50 Program against data races. 50 Program against datalocks. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely.			
25 Do not return null arrays or collections. 26 Do not call System.exit. 6 - Declarations and Statements 27 Have one concept per line. 28 Use braces in control structures. 29 Do not have empty blocks. 30 Use breaks in switch statements. 31 End switch statements with default. 32 Terminate if-else-if with else. 7 - Expressions 33 Restrict side effects in expressions. 34 Use named constants for non-trivial literals. 35 Make operator precedence explicit. 36 Do not use reference equality. 37 Use only short-circuit logic operators. 38 Do not use octal values. 39 Do not use floating point equality. 40 Use one result type in conditional expressions. 41 Do not use string concatenation operator in loops. 8 - Exceptions 42 Do not drop exceptions. 43 Do not abstript exist in finally block. 9 - Types 44 Use generics. 45 Use interfaces as types when available. 46 Use primitive types. 47 Do not remove literals from collections. 48 Restrict numeric conversions. 49 Program against data races. 50 Program against data races. 50 Program against data races. 50 Program against data races. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely.			
6 - Declarations and Statements 27 Have one concept per line. 28 Use braces in control structures. 29 Do not have empty blocks. 30 Use bracks in switch statements. 31 End switch statements with default. 7 - Expressions 33 Restrict side effects in expressions. 34 Use named constants for non-trivial literals. 35 Make operator precedence explicit. 36 Do not use reference equality. 37 Use only short-circuit logic operators. 38 Do not use result type in conditional expressions. 40 Use one result type in conditional expressions. 41 Do not use string concatenation operator in loops. 8 - Exceptions 42 Do not drop exceptions. 43 Do not drop exceptions. 44 Use generics. 45 Use generics. 46 Use primitive types. 47 Do not drop exceptions. 48 Restrict numeric conversions. 49 Program against data races. 50 Program against deadlocks. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely.			
Factor			
27 Have one concept per line. 28 Use braces in control structures. 29 Do not have empty blocks. 30 Use breaks in switch statements. 31 End switch statements with default. 32 Terminate if-else-if with else. 7 - Expressions 33 Restrict side effects in expressions. 34 Use named constants for non-trivial literals. 35 Make operator precedence explicit. 36 Do not use reference equality. 37 Use only short-circuit logic operators. 38 Do not use octal values. 39 Do not use floating point equality. 40 Use one result type in conditional expressions. 41 Do not use string concatenation operator in loops. 8 - Exceptions 42 Do not drop exceptions. 43 Do not abruptly exit a finally block. 9 - Types 44 Use generics. 45 Use interfaces as types when available. 46 Use primitive types. 47 Do not remove literals from collections. 48 Restrict numeric conversions. 10 - Concurrency 49 Program against data races. 50 Program against data races. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely.			
28 Use braces in control structures. 29 Do not have empty blocks. 30 Use bracks in switch statements. 31 End switch statements with default. 32 Terminate if-else-if with else. 7 - Expressions 33 Restrict side effects in expressions. 34 Use named constants for non-trivial literals. 35 Make operator precedence explicit. 36 Do not use reference equality. 37 Use only short-circuit logic operators. 38 Do not use rotal values. 39 Do not use floating point equality. 40 Use one result type in conditional expressions. 41 Do not use string concatenation operator in loops. 8 - Exceptions 42 Do not drop exceptions. 43 Do not abruptly exit a finally block. 9 - Types 44 Use generics. 45 Use interfaces as types when available. 46 Use primitive types. 47 Do not remove literals from collections. 48 Restrict numeric conversions. 10 - Concurrency 49 Program against data races. 50 Program against deadlocks. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely.			
29 Do not have empty blocks. 30 Use breaks in switch statements. 31 End switch statements with default. 32 Terminate if-else-if with else. 7 - Expressions 33 Restrict side effects in expressions. 34 Use named constants for non-trivial literals. 35 Make operator precedence explicit. 36 Do not use reference equality. 37 Use only short-circuit logic operators. 38 Do not use cotal values. 39 Do not use floating point equality. 40 Use one result type in conditional expressions. 41 Do not use string concatenation operator in loops. 8 - Exceptions 42 Do not drop exceptions. 43 Do not abruptly exit a finally block. 9 - Types 44 Use generics. 45 Use interfaces as types when available. 46 Use primitive types. 47 Do not remove literals from collections. 48 Restrict numeric conversions. 10 - Concurrency 49 Program against data races. 50 Program against data races. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely.			
30 Use breaks in switch statements. 31 End switch statements with default. 32 Terminate if-else-if with else. 7 - Expressions 33 Restrict side effects in expressions. 34 Use named constants for non-trivial literals. 35 Make operator precedence explicit. 36 Do not use reference equality. 37 Use only short-circuit logic operators. 38 Do not use octal values. 39 Do not use floating point equality. 40 Use one result type in conditional expressions. 41 Do not use string concatenation operator in loops. 8 - Exceptions 42 Do not drop exceptions. 43 Do not abruptly exit a finally block. 9 - Types 44 Use generics. 45 Use interfaces as types when available. 46 Use primitive types. 47 Do not remove literals from collections. 48 Restrict numeric conversions. 10 - Concurrency 49 Program against data races. 50 Program against data races. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely.			
31 End switch statements with default. 32 Terminate if-else-if with else. 7 - Expressions 33 Restrict side effects in expressions. 34 Use named constants for non-trivial literals. 35 Make operator precedence explicit. 36 Do not use reference equality. 37 Use only short-circuit logic operators. 38 Do not use octal values. 39 Do not use octal values. 39 Do not use ottal values. 40 Use one result type in conditional expressions. 41 Do not use string concatenation operator in loops. 8 - Exceptions 42 Do not drop exceptions. 43 Do not abruptly exit a finally block. 9 - Types 44 Use generics. 45 Use interfaces as types when available. 46 Use primitive types. 47 Do not remove literals from collections. 48 Restrict numeric conversions. 10 - Concurrency 49 Program against data races. 50 Program against data races. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely.			
7 - Expressions 33 Restrict side effects in expressions. 34 Use named constants for non-trivial literals. 35 Make operator precedence explicit. 36 Do not use reference equality. 37 Use only short-circuit logic operators. 38 Do not use octal values. 39 Do not use floating point equality. 40 Use one result type in conditional expressions. 41 Do not use string concatenation operator in loops. 8 - Exceptions 42 Do not drop exceptions. 43 Do not abruptly exit a finally block. 9 - Types 44 Use generics. 45 Use interfaces as types when available. 46 Use primitive types. 47 Do not remove literals from collections. 48 Restrict numeric conversions. 10 - Concurrency 49 Program against data races. 50 Program against data races. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely. 11 - Complexity			
7 - Expressions 33 Restrict side effects in expressions. 34 Use named constants for non-trivial literals. 35 Make operator precedence explicit. 36 Do not use reference equality. 37 Use only short-circuit logic operators. 38 Do not use octal values. 39 Do not use loating point equality. 40 Use one result type in conditional expressions. 41 Do not use string concatenation operator in loops. 8 - Exceptions 42 Do not drop exceptions. 43 Do not abruptly exit a finally block. 9 - Types 44 Use generics. 45 Use interfaces as types when available. 46 Use primitive types. 47 Do not remove literals from collections. 48 Restrict numeric conversions. 49 Program against data races. 50 Program against deadlocks. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely.			
33 Restrict side effects in expressions. 34 Use named constants for non-trivial literals. 35 Make operator precedence explicit. 36 Do not use reference equality. 37 Use only short-circuit logic operators. 38 Do not use octal values. 39 Do not use floating point equality. 40 Use one result type in conditional expressions. 41 Do not use string concatenation operator in loops. 8 - Exceptions 42 Do not drop exceptions. 43 Do not abruptly exit a finally block. 9 - Types 44 Use generics. 45 Use interfaces as types when available. 46 Use primitive types. 47 Do not remove literals from collections. 48 Restrict numeric conversions. 10 - Concurrency 49 Program against data races. 50 Program against deadlocks. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely.			
34 Use named constants for non-trivial literals. 35 Make operator precedence explicit. 36 Do not use reference equality. 37 Use only short-circuit logic operators. 38 Do not use octal values. 39 Do not use floating point equality. 40 Use one result type in conditional expressions. 41 Do not use string concatenation operator in loops. 8 - Exceptions 42 Do not drop exceptions. 43 Do not abruptly exit a finally block. 9 - Types 44 Use generics. 45 Use interfaces as types when available. 46 Use primitive types. 47 Do not remove literals from collections. 48 Restrict numeric conversions. 10 - Concurrency 49 Program against data races. 50 Program against data races. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely.			
35 Make operator precedence explicit. 36 Do not use reference equality. 37 Use only short-circuit logic operators. 38 Do not use octal values. 39 Do not use floating point equality. 40 Use one result type in conditional expressions. 41 Do not use string concatenation operator in loops. 8 - Exceptions 42 Do not drop exceptions. 43 Do not abruptly exit a finally block. 9 - Types 44 Use generics. 45 Use interfaces as types when available. 46 Use primitive types. 47 Do not remove literals from collections. 48 Restrict numeric conversions. 10 - Concurrency 49 Program against data races. 50 Program against data races. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely.			
36 Do not use reference equality. 37 Use only short-circuit logic operators. 38 Do not use octal values. 39 Do not use floating point equality. 40 Use one result type in conditional expressions. 41 Do not use string concatenation operator in loops. 8 - Exceptions 42 Do not drop exceptions. 43 Do not abruptly exit a finally block. 9 - Types 44 Use generics. 45 Use interfaces as types when available. 46 Use primitive types. 47 Do not remove literals from collections. 48 Restrict numeric conversions. 10 - Concurrency 49 Program against data races. 50 Program against datolocks. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely.			
37 Use only short-circuit logic operators. 38 Do not use octal values. 39 Do not use floating point equality. 40 Use one result type in conditional expressions. 41 Do not use string concatenation operator in loops. 8 - Exceptions 42 Do not drop exceptions. 43 Do not abruptly exit a finally block. 9 - Types 44 Use generics. 45 Use interfaces as types when available. 46 Use primitive types. 47 Do not remove literals from collections. 48 Restrict numeric conversions. 10 - Concurrency 49 Program against data races. 50 Program against deadlocks. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely. 11 - Complexity			
38 Do not use octal values. 39 Do not use floating point equality. 40 Use one result type in conditional expressions. 41 Do not use string concatenation operator in loops. 8 - Exceptions 42 Do not drop exceptions. 43 Do not abruptly exit a finally block. 9 - Types 44 Use generics. 45 Use interfaces as types when available. 46 Use primitive types. 47 Do not remove literals from collections. 48 Restrict numeric conversions. 10 - Concurrev 49 Program against data races. 50 Program against data races. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely.			
39 Do not use floating point equality. 40 Use one result type in conditional expressions. 41 Do not use string concatenation operator in loops. 8 - Exceptions 42 Do not drop exceptions. 43 Do not abruptly exit a finally block. 9 - Types 44 Use generics. 45 Use interfaces as types when available. 46 Use primitive types. 47 Do not remove literals from collections. 48 Restrict numeric conversions. 10 - Concurrency 49 Program against data races. 50 Program against deadlocks. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely.			
40 Use one result type in conditional expressions. 41 Do not use string concatenation operator in loops. 8 - Exceptions 42 Do not drop exceptions. 43 Do not abruptly exit a finally block. 9 - Types 44 Use generics. 45 Use interfaces as types when available. 46 Use primitive types. 47 Do not remove literals from collections. 48 Restrict numeric conversions. 10 - Concurrency 49 Program against data races. 50 Program against deadlocks. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely. 11 - Complexity			
41 Do not use string concatenation operator in loops. 8 - Exceptions 42 Do not drop exceptions. 43 Do not abruptly exit a finally block. 9 - Types 44 Use generics. 45 Use interfaces as types when available. 46 Use primitive types. 47 Do not remove literals from collections. 48 Restrict numeric conversions. 10 - Concurrency 49 Program against data races. 50 Program against deadlocks. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely. 11 - Complexity			
8 - Exceptions 42 Do not drop exceptions. 43 Do not abruptly exit a finally block. 9 - Types 44 Use generics. 45 Use interfaces as types when available. 46 Use primitive types. 47 Do not remove literals from collections. 48 Restrict numeric conversions. 10 - Concurrenty 49 Program against data races. 50 Program against deadlocks. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely. 11 - Complexity			
42 Do not drop exceptions. 43 Do not abruptly exit a finally block. 9 - Types 44 Use generics. 45 Use interfaces as types when available. 46 Use primitive types. 47 Do not remove literals from collections. 48 Restrict numeric conversions. 10 - Concurrenty 49 Program against data races. 50 Program against deadlocks. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely. 11 - Complexity		Do not use string concatenation operator in loops.	
 43 Do not abruptly exit a finally block. 9 - Types 44 Use generics. 45 Use interfaces as types when available. 46 Use primitive types. 47 Do not remove literals from collections. 48 Restrict numeric conversions. 10 - Concurrent 49 Program against data races. 50 Program against deadlocks. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely. 11 - Complexity 			
9 - Types 44 Use generics. 45 Use interfaces as types when available. 46 Use primitive types. 47 Do not remove literals from collections. 48 Restrict numeric conversions. 10 - Concurrenty 49 Program against data races. 50 Program against deadlocks. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely. 11 - Complexity			
44 Use generics. 45 Use interfaces as types when available. 46 Use primitive types. 47 Do not remove literals from collections. 48 Restrict numeric conversions. 10 - Concurrency 49 Program against data races. 50 Program against deadlocks. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely. 11 - Complexity		Do not abruptly exit a finally block.	
45 Use interfaces as types when available. 46 Use primitive types. 47 Do not remove literals from collections. 48 Restrict numeric conversions. 10 - Concurrency 49 Program against data races. 50 Program against deadlocks. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely. 11 - Complexity			
46 Use primitive types. 47 Do not remove literals from collections. 48 Restrict numeric conversions. 10 - Concurrency 49 Program against data races. 50 Program against deadlocks. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely. 11 - Complexity			
47 Do not remove literals from collections. 48 Restrict numeric conversions. 10 - Concurre— 49 Program against data races. 50 Program against deadlocks. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely. 11 - Complexity			
48 Restrict numeric conversions. 10 - Concurre— 49 Program against data races. 50 Program against deadlocks. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely. 11 - Complexity		Use primitive types.	
10 - Concurrency 49 Program against data races. 50 Program against deadlocks. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely. 11 - Complexity			
49 Program against data races. 50 Program against deadlocks. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely. 11 - Complexity			
50 Program against deadlocks. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely. 11 - Complexity	10 - Concurren		
50 Program against deadlocks. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely. 11 - Complexity			
51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely. 11 - Complexity	50	Program against deadlocks.	
52 Wait and notify safely. 11 - Complexity	51		
11 - Complexity	52		
	11 - Complexit		

