# 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

y outside the language definition. ith all warnings enabled; use static source code analyzers.  n  ble loop bounds for all loops meant to be terminating. direct or indirect recursion. dynamic memory allocation after task initialization. essages for task communication. task delays for task synchronization. ransfer write-permission (ownership) for shared data objects. ctions on the use of semaphores and locks. ry protection, safety margins, barrier patterns. goto, setjmp or longjmp. selective value assignments to elements of an enum list.  ta objects at smallest possible level of scope. return value of non-void functions, or explicitly cast to (void). validity of values passed to functions. and dynamic assertions as sanity checks. l16, etc instead of predefined C data types such as int, short, etc.		
ith all warnings enabled; use static source code analyzers.  n  ble loop bounds for all loops meant to be terminating.  direct or indirect recursion.  dynamic memory allocation after task initialization.  essages for task communication.  task delays for task synchronization.  ransfer write-permission (ownership) for shared data objects.  ctions on the use of semaphores and locks.  ry protection, safety margins, barrier patterns.  goto, setjmp or longjmp.  selective value assignments to elements of an enum list.  ta objects at smallest possible level of scope.  return value of non-void functions, or explicitly cast to (void).  validity of values passed to functions.  and dynamic assertions as sanity checks.  16, etc instead of predefined C data types such as int, short, etc.		
ble loop bounds for all loops meant to be terminating.  direct or indirect recursion.  dynamic memory allocation after task initialization.  essages for task communication.  task delays for task synchronization.  ransfer write-permission (ownership) for shared data objects.  ctions on the use of semaphores and locks.  ry protection, safety margins, barrier patterns.  goto, setjmp or longjmp.  selective value assignments to elements of an enum list.  ta objects at smallest possible level of scope.  return value of non-void functions, or explicitly cast to (void).  validity of values passed to functions.  and dynamic assertions as sanity checks.  16, etc instead of predefined C data types such as int, short, etc.		
ble loop bounds for all loops meant to be terminating.  direct or indirect recursion.  dynamic memory allocation after task initialization.  essages for task communication.  task delays for task synchronization.  ransfer write-permission (ownership) for shared data objects.  ctions on the use of semaphores and locks.  ry protection, safety margins, barrier patterns.  goto, setjmp or longjmp.  selective value assignments to elements of an enum list.  ta objects at smallest possible level of scope.  return value of non-void functions, or explicitly cast to (void).  validity of values passed to functions.  and dynamic assertions as sanity checks.  16, etc instead of predefined C data types such as int, short, etc.		
direct or indirect recursion.  dynamic memory allocation after task initialization. essages for task communication.  task delays for task synchronization. ransfer write-permission (ownership) for shared data objects.  ctions on the use of semaphores and locks. ry protection, safety margins, barrier patterns. goto, setjmp or longjmp.  selective value assignments to elements of an enum list.  ta objects at smallest possible level of scope. return value of non-void functions, or explicitly cast to (void). validity of values passed to functions. and dynamic assertions as sanity checks.  16, etc instead of predefined C data types such as int, short, etc.		
dynamic memory allocation after task initialization.  essages for task communication.  task delays for task synchronization.  ransfer write-permission (ownership) for shared data objects.  ctions on the use of semaphores and locks.  ry protection, safety margins, barrier patterns.  goto, setjmp or longjmp.  selective value assignments to elements of an enum list.  ta objects at smallest possible level of scope.  return value of non-void functions, or explicitly cast to (void).  validity of values passed to functions.  and dynamic assertions as sanity checks.  16, etc instead of predefined C data types such as int, short, etc.		
essages for task communication.  task delays for task synchronization.  ransfer write-permission (ownership) for shared data objects.  ctions on the use of semaphores and locks.  ry protection, safety margins, barrier patterns.  goto, setjmp or longjmp.  selective value assignments to elements of an enum list.  ta objects at smallest possible level of scope.  return value of non-void functions, or explicitly cast to (void).  validity of values passed to functions.  und dynamic assertions as sanity checks.  16, etc instead of predefined C data types such as int, short, etc.		
task delays for task synchronization.  ransfer write-permission (ownership) for shared data objects.  ctions on the use of semaphores and locks.  ry protection, safety margins, barrier patterns.  goto, setjmp or longjmp.  selective value assignments to elements of an enum list.  ta objects at smallest possible level of scope.  return value of non-void functions, or explicitly cast to (void).  validity of values passed to functions.  und dynamic assertions as sanity checks.  16, etc instead of predefined C data types such as int, short, etc.		
ransfer write-permission (ownership) for shared data objects.  ctions on the use of semaphores and locks.  ry protection, safety margins, barrier patterns.  goto, setjmp or longjmp.  selective value assignments to elements of an enum list.  ta objects at smallest possible level of scope.  return value of non-void functions, or explicitly cast to (void).  validity of values passed to functions.  und dynamic assertions as sanity checks.  16, etc instead of predefined C data types such as int, short, etc.		
ctions on the use of semaphores and locks.  ry protection, safety margins, barrier patterns.  goto, setjmp or longjmp.  selective value assignments to elements of an enum list.  ta objects at smallest possible level of scope.  return value of non-void functions, or explicitly cast to (void).  validity of values passed to functions.  und dynamic assertions as sanity checks.  16, etc instead of predefined C data types such as int, short, etc.		
ry protection, safety margins, barrier patterns. goto, setjmp or longjmp. selective value assignments to elements of an enum list.  ta objects at smallest possible level of scope. return value of non-void functions, or explicitly cast to (void). validity of values passed to functions. und dynamic assertions as sanity checks.  16, etc instead of predefined C data types such as int, short, etc.		
goto, setjmp or longjmp.  selective value assignments to elements of an enum list.  ta objects at smallest possible level of scope.  return value of non-void functions, or explicitly cast to (void).  validity of values passed to functions.  and dynamic assertions as sanity checks.  16, etc instead of predefined C data types such as int, short, etc.		
selective value assignments to elements of an enum list.  ta objects at smallest possible level of scope.  return value of non-void functions, or explicitly cast to (void).  validity of values passed to functions.  und dynamic assertions as sanity checks.  16, etc instead of predefined C data types such as int, short, etc.		
ta objects at smallest possible level of scope. return value of non-void functions, or explicitly cast to (void). validity of values passed to functions. and dynamic assertions as sanity checks.  16, etc instead of predefined C data types such as int, short, etc.		
return value of non-void functions, or explicitly cast to (void). validity of values passed to functions. und dynamic assertions as sanity checks.  16, etc instead of predefined C data types such as int, short, etc.		
return value of non-void functions, or explicitly cast to (void). validity of values passed to functions. und dynamic assertions as sanity checks.  16, etc instead of predefined C data types such as int, short, etc.		
validity of values passed to functions.  and dynamic assertions as sanity checks.  16, etc instead of predefined C data types such as int, short, etc.		
and dynamic assertions as sanity checks.  16, etc instead of predefined C data types such as int, short, etc.		
16, etc instead of predefined C data types such as int, short, etc.		
rder of evaluation in compound expressions explicit.		
expressions with side effects.		
4 Code Clarity		
very limited use of the C pre-processor.		
ine macros within a function or a block.		
efine or redefine macros.		
, #elif, and #endif in the same file as the matching #if or #ifdef.		
ore than one statement or declaration per line of text.		
functions with a limited number of parameters.		
re than two levels of indirection per declaration.		
re than two levels of dereferencing per object reference.		
e dereference operations inside macros or typedefs.		
non-constant function pointers.		
function pointers into other types.		
ce code or declarations before an #include directive.		
iance		
a shall rules not already covered at Levels 1-4.		
6 - MISRA should compliance		
pliance		
1		

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	B – JPL Coding Standard for Java		
2 Apply static analysis 3 Document public elements. 4 Write unit tests. 2 - Names 5 Use the standard naming conventions 6 Do not override field or class names. 3 - Packages. Classes and Interfaces 7 Make improte explicit. 8 Do not have cyclic package dependencies. 9 Obey the contract for ciguidy. 11 Define both equality and hashCode). 11 Define cquals when adding fields. 12 Define cquals when adding fields. 13 Do not see final their Cloneabile interface. 14 Define cquals when adding fields. 15 Do not call nontrinal methods in constructors. 16 Select composition over inheritance. 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 annotations. 23 Restrict method overloading. 24 Do not assign parameters. 25 Do not return unit arrays or collections. 26 Do not easily subserne exit. 27 Maye one concept per fine. 28 Do not assign parameters. 29 Use because in the control structures. 29 Use annotations. 30 Do not all system exit. 31 End systich statements 32 Use parameters. 33 Restrict method overloading. 34 Do not assign parameters. 35 Do not call System exit. 36 De not call system exit. 37 Expressions. 38 De not assign provide control structures. 39 De not call system exit. 30 Use assertions and Statements 31 End systich statements with default. 31 End systich statements with default. 32 I Terminat it riches if with else. 34 Use non-control structures. 35 Make operator precience equality. 36 Do not assign providence control structures. 39 Do not one swell type in conditional expressions. 31 End systich statements with default. 32 I Terminate it riches if with else. 34 Use name control structures. 35 De not easily stopped concentrol into-private literals. 36 De not see cell values. 37 De not easily stopped concentrol into-private literals. 38 Do not use the control into-private literals. 39 Do not use the control into-private literals. 40 De not resolve the seed of the surface of the surface of the surface of	1 – Process		
2 Apply static analysis 3 Document public elements. 4 Write unit tests. 2 - Names 5 Use the standard naming conventions 6 Do not override field or class names. 3 - Packages. Classes and Interfaces 7 Make improte explicit. 8 Do not have cyclic package dependencies. 9 Obey the contract for ciguidy. 11 Define both equality and hashCode). 11 Define cquals when adding fields. 12 Define cquals when adding fields. 13 Do not see final their Cloneabile interface. 14 Define cquals when adding fields. 15 Do not call nontrinal methods in constructors. 16 Select composition over inheritance. 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 annotations. 23 Restrict method overloading. 24 Do not assign parameters. 25 Do not return unit arrays or collections. 26 Do not easily subserne exit. 27 Maye one concept per fine. 28 Do not assign parameters. 29 Use because in the control structures. 29 Use annotations. 30 Do not all system exit. 31 End systich statements 32 Use parameters. 33 Restrict method overloading. 34 Do not assign parameters. 35 Do not call System exit. 36 De not call system exit. 37 Expressions. 38 De not assign provide control structures. 39 De not call system exit. 30 Use assertions and Statements 31 End systich statements with default. 31 End systich statements with default. 32 I Terminat it riches if with else. 34 Use non-control structures. 35 Make operator precience equality. 36 Do not assign providence control structures. 39 Do not one swell type in conditional expressions. 31 End systich statements with default. 32 I Terminate it riches if with else. 34 Use name control structures. 35 De not easily stopped concentrol into-private literals. 36 De not see cell values. 37 De not easily stopped concentrol into-private literals. 38 Do not use the control into-private literals. 39 Do not use the control into-private literals. 40 De not resolve the seed of the surface of the surface of the surface of	1	Compile with checks turned on	
3 Documen public elements 4 Write unit tests. 2 - Names 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 controlled year of the control structure. 11 Define equals when adding fields. 12 Define equals when adding fields. 12 Define equals when adding fields. 13 Do not sus finalizers. 14 Do not replace the control structure. 15 Do not call nonfinal methods in constructors. 16 Select composition over inheritance. 17 Make fields private. 18 Do not see state metable fields final. 19 Define equals when adding fields. 19 Do not see state metable fields final. 19 Do not see state metable fields final. 10 Do not see state metable fields final. 11 Use assertions. 12 Use assertions. 12 Use assertions. 12 Use assertions. 13 Do not exist multi- fields before use. 14 Do not assign to parameters. 15 Do not return unit arrays or collections. 16 Do not return unit arrays or collections. 17 Use one concept per line. 18 Do not assign to parameters. 19 Do not have empty blocks. 10 Do not all System exit. 10 Do not all System exit. 10 Do not all System exit. 11 But switch statements with default. 12 Do not as ereference equality. 13 Do not use reference equality. 14 Do not use reference equality. 15 Do not even the proceedence explicit. 16 Do not use reference equality. 17 Do not even the proceedence explicit. 18 Do not use reference equality. 19 Do not even the proceedence explicit. 20 Do not even the proceedence explicit. 21 Do not even the proceedence explicit. 22 Do not even the proceedence explicit. 23 Do not use reference equality. 24 Do not even the even the scheduler for	2		
4 Write unif tests  7 Names  5 Use the standard naming conventions. 6 Do not override field or class manes  3 - PacKagos, Classos and Interfaces  8 Do not averride field or class manes  9 Obey the control of the cont			
2 - Names			
5 Use the standard naming conventions. 6 Do not override field or class manes. 3 - Packages, Classes and Interfaces  7 Make imports explicit. 8 Do not have eyelic package dependencies. 9 Obey the contract for equals(). 10 Define pulsa when adding fields. 11 Define equals when adding fields. 12 Define equals with parameter type Object. 13 Do not use finalizers. 14 Do not use finalizers. 14 Do not use finalizers. 15 Do not call malizers. 16 Select composition over inheritance. 17 Make fields private. 18 Do not use finalin almethods in constructors. 18 Do not use strain and the fields. 19 Define equals be finalizers. 19 Define equals with parameter type Object. 10 Initiative fields before use. 10 Initiative fields before use. 20 Initiative fields before use. 21 Use ansorations. 22 Use annotations. 23 Restrict method overloading. 24 Do not all system exit. 25 Do not relative interfaces. 26 Do not relative interfaces. 27 Illus one concept per line. 28 Use braces in control structures. 29 Do not have empty blocks. 30 Use braces in such distinctions. 31 Individe statistics and structures. 32 Restrict method in the control structures. 33 Restrict side effects in expressions. 34 Per parameters. 35 Do not relative interfaces. 36 Do not expressions. 37 Expressions. 38 Restrict side effects in expressions. 39 Restrict side effects in expressions. 30 Use on the control structures. 31 Use on the control structures. 32 Do not have empty blocks. 33 Use on the control structures. 34 Do not always the statements with default. 35 Do not expression the statements with default. 36 Do not expression the statements with default. 37 Expressions. 38 Petron studies and constants for non-enviral literals. 39 Do not have empty blocks. 30 Use braces in such distributional expressions. 31 Use only short-circuit leptoperators. 32 Do not always they should be such as a superational expressions. 39 Do not use from the such definitional expressions. 40 Do not expression the such definitional expressions. 41 Do not report blictures from collecti			
6 Do not override field or class names.  7 Akales, Classes and Interfaces  7 Make imports explicit.  8 Do not have evely package dependencies.  9 Obey the contract for equals().  10 Define both equals() and hash Code().  11 Define capuals when adding fields.  12 Define capuals when adding fields.  13 Do not use finalizers.  14 Do not use finalizers.  14 Do not call nonfinal methods in constructors.  15 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 - Methods  21 Use assertions.  22 Use annotations.  23 Restrict method overloading.  24 Do not actum null arrays or collections.  25 Do not return null arrays or collections.  26 Do not return null arrays or collections.  27 Do not even ply blocks.  30 Use breaks in switch statements.  4 Have one concept per line.  27 Have one concept per line.  28 Use haves in control structures.  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 floating point equality.  47 Use one result year in control structures.  39 Do not use floating point equality.  40 Use one result year in control structures.  41 Do not use string concatenation operator in loops.  42 Do not drop exceptions.  43 Do not the refuse capacity exit in a finally block.  44 Use generies.  45 De one the switch statements.  46 De Greater of the switch statements.  47 De one true with every exit in any pressions.  48 Exceptions  49 Do not the refuse conversions.  40 Do not the refuse conversions.  41 Do not the refuse conversions.  42 Do not drop exceptions.  43 Do not the refuse conversions.  44 Use generies.  45 Define capacity with every exit finally block.  9 - Tygens against data races.  9 Program against data races.  9			
3 - Packages, Classes and Interfaces 7 N Make imports explicit 8 Do not have explic package dependencies 9 Obey the contract for equals() and hashCode(). 11 Define equals when adding fields. 12 Define equals when adding fields. 13 Do not use inflaitzers. 14 Do not implement the Cloneable interface. 15 Do not call minifiam dimbods in constructors. 16 Select composition over inheritance. 17 Make fields private. 18 Do not are stain matable fields. 19 De note in minifiam dimbods in constructors. 20 Initiative fields before use. 21 Initiative fields before use. 22 Use assertions. 23 Restrict method overloading. 24 Do not discipate to parameters. 25 Do not cut return half arrays or collections. 26 Do not all experiments. 27 Do not cut return half arrays or collections. 28 Do not all experiments. 29 Do not all experiments. 29 Do not all experiments. 20 Do not all experiments. 21 Tail and switch statements. 22 Do not all experiments with default. 23 It and switch statements. 34 Use mand constants for non-trivial literals. 35 Restrict side effects in expressions. 36 Do not use Intendence explicit. 37 Expressions 38 Do not use Indiance of the processions. 39 Do not use Indiance of the processions. 30 Use breaks in switch statements with default. 31 Do not use Indiance of the processions. 32 Use and constants for non-trivial literals. 33 Do not use Indiance of the processions. 34 Use named constants for non-trivial literals. 36 Do not use Indiance of the processions. 37 Use only short-circuit logic operators. 38 Do not use Indiance of the processions. 39 Do not not be entire the safety with a finally block. 30 Do not the entire conversions. 41 Use pagentic deallocks. 42 Do not drove on the sc			
7 Make imports explicit. 8 Do not have eyelic package dependencies. 9 Obey the contract for equals(). 10 Define both equals() and hashCode(). 11 Define counts when adding fields. 12 Define equals when adding fields. 13 De not use finalizers. 14 De 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 De not use static mutable fields. 19 Do not use static mutable fields. 19 Do not use static mutable fields. 20 Initialize fields before use. 21 Use assertions. 22 Use amoutations. 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 Do not return unit arrays or collections. 28 Do not even unit arrays or collections. 29 Do not even unit arrays or collections. 20 Do not call System exit. 20 Do not even unit arrays or collections. 21 Do not return unit arrays or collections. 22 Do not return unit arrays or collections. 25 Do not return unit arrays or collections. 26 Do not even switch statements 27 Have one concept per line. 28 Use braces in switch statements. 30 Use bracks in switch statements. 31 End switch statements with default. 32 Terminate refersed with desault. 33 Terminate refersed with desault. 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 reference equality. 40 Use one result type in conditional expressions. 41 Use generies. 42 Use not result type in conditional expressions. 43 Use named constants for non-trivial literals. 44 Use generies. 45 Use interfaces as types when available. 46 Use privative types. 47 Do not remove literals from collections. 48 Use privative conditional expressions. 49 Pogram against deatlocks. 51 Do not not work the scheduler for synchronization.			
8 Do not have eyelic package dependencies. 9 Obey the contract for equals?) 10 Define both equals () and hashCode() 11 Define capals when adding fields. 12 Define capals when adding fields. 13 De not use finalizers. 14 Do not implement the Cloneable interface. 15 Do not call minifiam embrods in constructors. 16 Select composition over inheritance. 17 Make fields private. 18 Do not are state mutable fields. 19 De not are state mutable fields. 19 De not are state mutable fields. 19 Declare immutable fields final. 10 District in the fields private. 19 Declare immutable fields final. 10 Declare immutable fields final. 10 Declare immutable fields final. 11 Seasestions. 22 Use amotations. 23 Restrict method overloading. 24 Do not assign to parameters. 25 Do not return and parameters. 26 Do not return and ary sor collections. 27 Do not call system; exit. 28 Do not return and ary sor collections. 29 Do not return and arstements. 20 Do not return and arstements. 21 Takey one concept per line. 22 Use breaks in switch statements. 23 Use breaks in countof structures. 24 Do not also septicated the fields. 25 Do not return and arstements. 27 Texpressions. 28 Do not also seption before the fields. 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. 33 Restrict side effects in expressions. 34 Use anamed constants for non-rivial literals. 35 Make operator precedence explicit. 36 Do not use Inding point equality. 37 Use only short-circuit logic operators. 38 Do not use Inding point equality. 39 Do not not be empty blocks. 40 Use one result type in conditional expressions. 41 Use one result type in conditional expressions. 42 Use not not well water. 43 Use primative pain expensions. 44 Use generies. 45 Use primative pain expensions. 46 Do not not reflex type in conditional expressions. 47 Use only short-circuit logic operators. 48 Use primative pain expensions. 49 Do not remove literals from collections. 41 Use primative conversions. 51 Do not	3 - Packages, 0		
9 Obey the contract for equals() 10 Define both oquals() 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 De not use static mutable fields. 19 De not use static mutable fields final. 10 Initialize fields before use 5 - Mothods 21 Use assertions. 22 Use annotations. 23 Restrict method overloading, 24 De not assign to parameters. 25 De not return util arrays or collections. 26 De not call System exit 6 - Declarations and Statements 27 Iliave one concept per line. 28 Use braces in control structures. 29 De not have empty blocks. 30 Use bracks in switch statements with default. 31 End switch statements with default. 32 Terminate if else-if-witch for non-rivial literals. 33 Restrict side effects in expressions. 34 Use named constants for non-rivial literals. 35 De not use return prededence equality. 36 De not use return prededence explicit. 37 Use only short-circuit logic operations. 39 De not use floating point equality. 30 Use bracks in Switch statements with default. 31 Use named constants for non-rivial literals. 32 De not use reference equality. 33 De not use reference equality. 34 Use named constants for non-rivial literals. 35 De not use reference equality. 46 Use primitive types when available. 47 De not use string concatenation operator in loops. 48 Execeptions. 49 De not true from gainst deatlocks. 50 De program against deatlocks. 51 De not retrow the scheduler for synchronization. 52 Use not short one scheduler for synchronization. 53 De not trend the scheduler for synchronization. 54 De not trend the scheduler for synchronization. 55 De not trend the scheduler for synchronization.	7	Make imports explicit.	
10   Define both equals() and hashCode().   11   Define equals with parameter type Object.   12   Define equals 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.   20   Initialize fields before use.   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 return null arrays or collections.   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   Zerminate Flessel-F with else.   34   Use named constants for non-trivial literals.   35   Make operator precedence equality.   37   Use only short-circuit logic operators.   39   Do not use reference equality.   31   Use no to sub-circuit logic operators.   32   Do not not use reference equality.   37   Use only short-circuit logic operators.   39   Do not use reference equality.   37   Use only short-circuit logic operators.   39   Do not use reference equality.   30   Use not short-circuit logic operators.   31   Do not use reference equality.   32   Use not short-circuit logic operators.   34   Use not short-circuit logic operators.   35   Do not alone position operator in loops.   4   Use one result type in conditional expressions.   5   Denot alone to the scheduler for synchronization.   4   Use one type to the sched	8	Do not have cyclic package dependencies.	
10   Define both equals() and hashCode().   11   Define equals with parameter type Object.   12   Define equals 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.   20   Initialize fields before use.   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 return null arrays or collections.   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   Zerminate Flessel-F with else.   34   Use named constants for non-trivial literals.   35   Make operator precedence equality.   37   Use only short-circuit logic operators.   39   Do not use reference equality.   31   Use no to sub-circuit logic operators.   32   Do not not use reference equality.   37   Use only short-circuit logic operators.   39   Do not use reference equality.   37   Use only short-circuit logic operators.   39   Do not use reference equality.   30   Use not short-circuit logic operators.   31   Do not use reference equality.   32   Use not short-circuit logic operators.   34   Use not short-circuit logic operators.   35   Do not alone position operator in loops.   4   Use one result type in conditional expressions.   5   Denot alone to the scheduler for synchronization.   4   Use one type to the sched	9	Obey the contract for equals().	
11   Define equals when adding fields.	10		
12   Define equals with parameter type Object.			
13 Do not use finalizers 14 Do not implement the Clonable interface. 15 Do not call nonfinal methods in constructors. 16 Select composition over inheritance.  4 - Fields 17 Make fields private. 18 Do not use state mutable fields. 19 Do clare immutable fields final. 20 Initializer fields before use. 5 - Methods 21 Use assertions. 22 Use annotations. 23 Restrict method overloading. 24 Do not assign to parameters. 25 Do not return util arrays or collections. 26 Do not call System.exit. 27 Do not call System.exit. 28 Do not call System.exit. 29 Do not shave empty blocks. 30 Use braces in control structures. 29 Do not the empty blocks. 30 Use bracks in switch statements. 31 End switch statements with default. 32 Terminate if-else-off with else. 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 reference equality. 39 Do not survey as on the quality. 40 Use one result type in conditional expressions. 41 Do not use reference equality. 42 Do not day one preference explicit. 43 Do not use reference equality. 44 Use generics. 45 Do not day one preference explicit. 46 Do not survey in conditional expressions. 47 Do not care locating point equality. 48 Do not use reference explicit from collections. 49 Do not survey in conditional expressions. 41 Do not use reference explicit. 45 Do not explain the quality. 46 Use generics. 47 Do not care locating point equality. 48 Estrict numeric conversions. 49 Do not day one presult type in conditional expressions. 41 Do not use reference equality on the present of the pression of the pressi			
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. 20 Initialize fields before use. 5 - Methods 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. 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. 37 - Expressions 38 Restrict side effects in expressions. 39 Use non-instructive in control structures. 30 Do not use or content per line. 31 Use annotation of the statements with default. 32 Terminate if-else-if with else. 33 Restrict side effects in expressions. 34 Use named constants for non-trivial literals. 35 Do not use floating point equality. 36 Do not use reference equality. 37 Use only short-circuit logic operators. 38 Do not use floating point equality. 40 Use one result type in conditional expressions. 41 Do not drop exceptions. 42 Do not drop exceptions. 43 Do not use string concatenation operator in loops. 44 Use generies. 45 Use interfaces as types when available. 46 Use primitive types. 47 Do not remove literals from collections. 48 Restrict numeric conversions. 59 Program against deallocks. 51 Do not vit und notify safely. 51 Do not rely on the scheduler for synchronization. 50 Do not use on the one of the program against deallocks. 51 Do not vit und notify safely.			
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. 20 Initialize fields before use.  5 - Methods  21 Use assertions. 22 Use amontations. 23 Restrict method overloading. 24 Do not assign to parameters. 25 Do not return null arrays or collections. 26 Do not return null arrays or collections. 27 Have one concept per line. 28 Use braces in control structures. 29 Do not have empty blocks. 30 Use braces in switch statements. 31 End switch statements with default. 31 End switch statements with default. 32 Terminate fields-eff with else. 33 Restrict side effects in expressions. 34 Use anamed 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 string concatenation operator in loops. 49 Do not have the value of the string			
4 - Fields 17 Make fields private. 18 Do not use static mutable fields. 19 Declare immutable fields final. 20 Initialize fields before use. 5 - Methods 21 Use annotations. 221 Use annotations. 222 Use annotations. 223 Restrict method overloading. 244 Do not assign to parameters. 25 Do not return null arrays or collections. 26 Do not all System exit. 6 - Declarations and Statements 27 Have one concept per line. 28 Use braces in control structures. 29 Do not all system exit. 30 Use breaks in switch statements. 31 End switch statements with default. 32 Terminate li-fist-set 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 foliating point equality. 49 Use one stult ype in conditional expressions. 41 Do not sure freence equality. 40 Use one result type in conditional expressions. 41 Use named constants for non-trivial literals. 42 Do not use freence equality. 43 Do not use foliating point equality. 44 Use one result type in conditional expressions. 45 Exceptions 46 Use primitive types. 47 Do not remove literals from collections. 48 Restrict numeric conversions. 49 Program against deatlocks. 51 Do not ut on the scheduler for synchronization. 52 Wat and notify safely.			
17			
17 Make fields private 18 Do not use static mutable fields. 19 Declare immutable fields final. 20 Initialize fields before use. 5 - Methods 21 Use annotations. 22 Use annotations. 23 Restrict method overloading. 24 Do not assign to parameters. 25 Do not return mull arrays or collections. 26 Do not assign to parameters. 27 Do not assign to parameters. 28 Do not all System exit. 29 Do not assign to parameters. 20 Do not assign to parameters. 21 Have one concept per line. 22 Use breass in control structures. 23 Do not have empty blocks. 30 Use breass in control structures. 31 End switch statements with default. 32 Terminate if-else-if with else. 33 Terminate if-else-if with else. 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 reference equality. 40 Use one result type in conditional expressions. 41 Do not use string concatenation operator in loops. 42 Do not use string concatenation operator in loops. 43 Exceptions 44 Use generies. 45 Use interfaces as types when available. 46 Use primitive types. 47 Do not new or literals from collections. 48 Restrict numeric conversions. 49 Program against data races. 50 Program against data races. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely.			
18			
18	17	Make fields private.	
5 - Methods 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 assign to parameters. 27 Do not assign to parameters. 28 Do not assign to parameters. 29 Do not return null arrays or collections. 29 Do not return null arrays or collections. 30 Use braces in control structures. 30 Use braces in control structures. 31 End switch statements with default. 32 Terminate if-else-if with else. 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 reference equality. 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 44 Use generics. 45 Use interfaces as types when available. 46 Use primitive types. 47 Do not termove literals from collections. 48 Restrict numeric conversions. 49 Program against data races. 50 Do not rely on the scheduler for synchronization.	18		
S - Methods	19	Declare immutable fields final.	
S - Methods	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 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 reference equality. 40 Use one result type in conditional expressions. 41 Do not use string concatnation operator in loops. 42 Do not drop exceptions. 43 Do not use string concatnation operator in loops. 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 deallocks. 51 Do not tely on the scheduler for synchronization. 52 Wait and notify safely. 11 - Complexity	5 - Methods		
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 refore no explicit. 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 as string concatenation operator in loops. 8 - Exceptions 44 Use generies. 45 Use generies. 46 Use printive types. 47 Do not remove literals from collections. 48 Restrict numeric conversions. 49 Program against deadlocks. 50 Program against deadlocks. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely. 51 Complexity		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 octal values. 39 Do not use otal values. 39 Do not use otal values. 39 Do not use string concatenation operator in loops.  8 - Exceptions  41 Do not drop exceptions. 42 Do not drop exceptions. 43 Do not abruptly exit a finally block. 9 - Types  44 Use generics. 45 Use primitive types. 46 Use primitive types. 47 Do not remove literals from collections. 48 Restrict numeric conversions. 40 Program against data races. 50 Program against data faces. 50 Program against data faces. 50 Program against data faces. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely.			
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 reference 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 data faces. 50 Program against data faces. 50 Program against data forces. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely.			
Do not return null arrays or collections. Do not call System.exit. 6 - Declarations and Statements  127   Have one concept per line. 128   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 otal 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 drop exceptions. 44   Use generies. 45   Use primitive type sin conditional expressions. 46   Use primitive type sin conditional expressions. 47   Do not drop exceptions. 48   Use generies. 49   On ont drop exceptions. 40   Use generies. 41   Use generies. 42   Use primitive types in conditional expressions. 43   Do not the primitive types. 44   Use generies. 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 faces.			
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.  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 drop exceptions.  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.		E I	
A			
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 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 abstruptly exit a finally block. 9 - Types 44 Use generics. 45 Use interfaces as types when available. 46 Use primitive types. 47 Do not emove 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 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 otal 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.	6 - Declaration	s and Statements	
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. 49 Program against data races. 50 Program against data races. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely.	27	Have one concept per line.	
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. 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.	
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 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 aboutly 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. 51 - Complexity	29	Do not have empty blocks.	
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.  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 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 deadlocks.  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 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 deadlocks. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely.  11 - Complexity			
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 deadlocks. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely.  11 - Complexity  11 - Complexity			
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 deallocks. 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 data for synchronization. 52 Wait and notify safely.  11 - Complexity			
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 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 - Concurrency 49 Program against data races. 50 Program against deadlocks. 51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely.	36		
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	37	Use only short-circuit logic operators.	
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.	
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	39	Do not use floating point equality.	
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 - 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			
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		Do not use string concurention operator in toops.	
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	•	Do not doon avanations	
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			
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 - 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			
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		
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		
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			
51 Do not rely on the scheduler for synchronization. 52 Wait and notify safely. 11 - Complexity			
52 Wait and notify safely.  11 - Complexity			
11 - Complexity			
53 Reduce code complexity.			
	53	Reduce code complexity.	