

SISSy: Catalog of Detected Problem Patterns

Adrian Trifu, Mircea Trifu
FZI Forschungszentrum Informatik
Karlsruhe, Germany

Table of contents

| | | |
|-------|--|----|
| 1. | Introduction..... | 4 |
| 2. | Catalog of detected structural problems..... | 5 |
| 2.1. | Attribute overlap..... | 5 |
| 2.2. | Complex method | 6 |
| 2.3. | Constant redefinition | 6 |
| 2.4. | Cyclical dependency between classes | 6 |
| 2.5. | Cyclical dependency between packages..... | 7 |
| 2.6. | Cyclical dependency between source files | 7 |
| 2.7. | Cyclical method calls | 7 |
| 2.8. | Dead attribute | 8 |
| 2.9. | Dead code | 8 |
| 2.10. | Dead imports | 8 |
| 2.11. | Dead method | 9 |
| 2.12. | Duplicated code..... | 9 |
| 2.13. | General parameter | 9 |
| 2.14. | Generation conflict | 10 |
| 2.15. | God class, attribute form | 10 |
| 2.16. | God class, method form..... | 10 |
| 2.17. | God file..... | 11 |
| 2.18. | God method | 11 |
| 2.19. | God package..... | 11 |
| 2.20. | Ignored abstraction | 12 |
| 2.21. | Import chaos | 12 |
| 2.22. | Improper name length | 13 |
| 2.23. | Inappropriate commenting..... | 13 |
| 2.24. | Inconsistent operations | 13 |
| 2.25. | Informal documentation | 14 |
| 2.26. | Interface-Bypass | 14 |
| 2.27. | Knows of derived | 15 |
| 2.28. | Late abstraction | 15 |
| 2.29. | Long parameter list..... | 15 |
| 2.30. | Mini-classes..... | 16 |
| 2.31. | Mini-packages | 16 |
| 2.32. | Misleading naming of source files..... | 16 |
| 2.33. | Object placebo, attribute form | 17 |
| 2.34. | Object placebo, method form | 17 |
| 2.35. | Orphan sibling attributes | 17 |
| 2.36. | Orphan sibling methods..... | 18 |
| 2.37. | Overloaded file | 18 |
| 2.38. | Permissive visibility, attribute form | 18 |
| 2.39. | Permissive visibility, method | 19 |
| 2.40. | Polymorphic calls in constructor | 19 |
| 2.41. | Polymorphism placebo | 19 |
| 2.42. | Refused bequest, implementation form | 20 |
| 2.43. | Refused bequest, interface form | 20 |
| 2.44. | Reversed package and inheritance hierarchies | 21 |
| 2.45. | Risky code | 21 |
| 2.46. | Similar unrelated abstractions | 21 |
| 2.47. | Simulated polymorphism..... | 22 |
| 2.48. | Type name duplication | 22 |
| 2.49. | Unfinished code..... | 22 |
| 2.50. | Variables having constant value..... | 23 |

| | | |
|-------|---------------------------------------|----|
| 2.51. | Violation of data encapsulation | 23 |
| 2.52. | Violation of naming convention | 23 |

1. Introduction

The tool for structural investigation of software systems (SISSy) is an open-source platform for the automated detection of structural flaws. It was designed to be integrated in the build process in order to regularly provide reports on the internal quality of the developed system. If in the course of development, problems arise in the structure, they are immediately identified and reported, giving developers the opportunity to fix them before they get unmanageable.

SISSy has its origins in the tool chain GOOSE, which has been developed at FZI between 1998 and 2005 for the languages Java and C++. GOOSE had a simple relational model at its core, which could be persisted in the form of Prolog facts. Using a Prolog inference machine, the fact repository could then be queried in order to compute various metrics as well as search for basic structural flaws.

SISSy is not merely a further development of GOOSE, but a completely new design. The model at the core of SISSy is much more fine-grained, and allows for much more complex analyses to be performed.

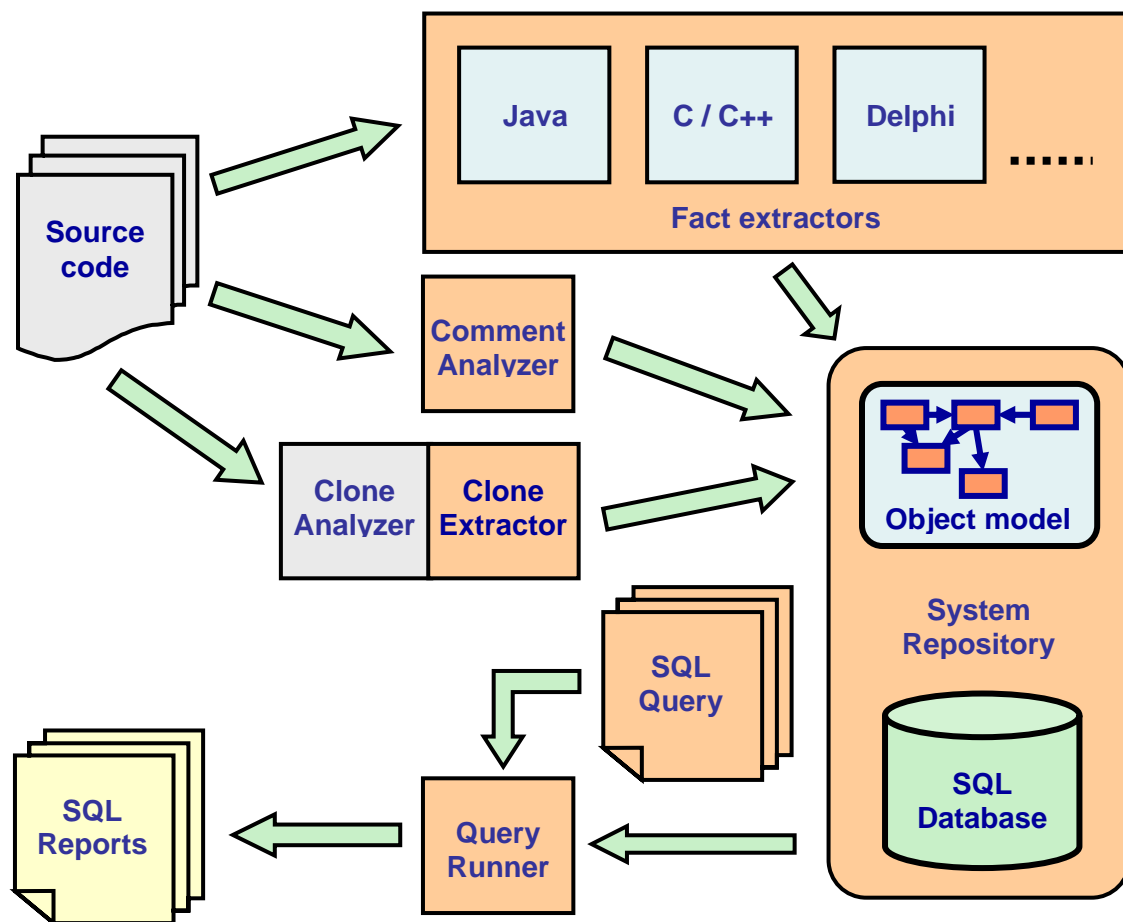


Figure 1: Architecture of SISSy

Apart from simple metrics such as lines of code, number of classes, methods, attributes, etc, SISSy is also able to compute complex metrics such as depth of an inheritance tree, cyclomatic complexity and various sorts of coupling and cohesion. Furthermore, based on heuristics, it is able to automatically analyze the system and find instances of code smells and various other violations of accepted principles and rules of design. And this is not all. SISSy also implements the basic technology for simulating code transformations (refactorings) on its abstract model, which makes impact analysis, problem correction and computing various optimizations possible.

Figure 1 illustrates the architecture of Sissy. The central part of the tool is the system repository, an abstract, language independent model of the source code. The repository has two forms of representation: as an object model and as an SQL database, which are complementary to each-other. On one hand, the object model is a very flexible, dynamically transformable representation, providing a rich API, that is best suited for impact analyses and optimizations. It is capable of simulating most of the known refactorings and their effects while guaranteeing the syntactical correctness of the model. On the other hand, the SQL database is a persistent and easy to query representation that is optimal for analysis involving very large data sets.

The three fact extractors are responsible for constructing the object model representation of the system from its source code, by mapping the various language specific meta-models onto the language independent meta-model of the system repository. Each fact extractor deals with one of the supported programming languages (currently Java, C++ and Delphi).

The CloneExtractor and CommentAnalyzer are two modules that are responsible for enriching the object model with information concerning blocks of duplicated code, as well as comments.

CloneExtractor uses an external open-source tool called CloneAnalyzer, which performs text based detection of duplicated code fragments. Once identified, the text-based clones are consequently mapped onto the abstract instructions already contained in the object model.

The module CommentAnalyzer is responsible for extracting and classifying source code comments based on a set of typical user-definable patterns, such as ToDos, FixMes, Hacks, etc. A further responsibility of the module is to annotate the relevant object model entities with the previously determined comment information.

Finally, the module called Query Runner is responsible for carrying out the SQL queries for problem detection in a batch mode, as well as for generating various reports.

Sissy defines a number of 52 highly optimized queries that are intended for everyday execution. They cover the entire range of code smells and various design principles and rules from the architectural down to the code level. A complete list of detected problems can be found in section 2.

Typical use-cases for Sissy are the detailed analysis of a system, daily analysis which is integrated with the build process, and trend analysis.

The detailed analysis is typically performed by external specialists and can be used to assess the current state of the system. It is goal oriented, and typically occurs right after a release or acceptance of a code base from a subcontractor.

The daily analysis that is integrated in the build process is useful for detecting and reporting problems to the relevant developers on a short notice (i.e. during development).

Finally, a periodical trend analysis is especially useful for project managers that want to recognize potential maintenance risks as early as possible, in order to be able to make effective decisions concerning corrective measures. A one to two week period between consecutive runs is recommended.

To this date, Sissy has been successfully employed in analyzing numerous open-source as well as commercial Systems of up to 2 million lines of code, from various domains such as engineering, telecommunications, billing, logistic, insurance, embedded systems, CRM and research. These systems employ a very wide palette of infrastructure technologies as well as all three currently supported programming languages (Java, C++ and Delphi).

2. Catalog of detected structural problems

This section contains a list of the standard problems, currently detectable in Sissy. For each problem, we provide a short description.

2.1. Attribute overlap

Definition

The name of a non-static attribute of a superclass is reused for a different attribute in one of the direct or indirect subclasses of the superclass. Type and visibility of the attributes is not relevant.

Motivation

If this attribute is referenced from a client, it covers the attribute defined in the superclass (provided it is visible there). However in theory this may be the desired effect, it brings additional unclarity to the code, which increases the risk of making mistakes during maintenance.

Languages

C++, Java, C#, Delphi

2.2. Complex method

Definition

A method having the cyclomatic complexity value higher than 10. The value is computed as the number of control flow altering statements plus 1. Control flow altering statements are if, else, case and all loops.

Motivation

The more different possible execution paths a method has, the harder it is to understand and change it. Furthermore writing tests for such methods is also more difficult.

Languages

C++, Java, C#, Delphi

2.3. Constant redefinition

Definition

A global constant (e.g. public static final, public static const) is defined more than once throughout the system.

Motivation

If global constants are defined in several places, there is a danger that the maintainer/developer changes one definition but forgets to update another. This may cause runtime failures. On the other hand, if the constants have different semantics, they should also have different names. Otherwise the understandability of the code using them has a lot to suffer.

Languages

C++, Java, C#, Delphi

2.4. Cyclical dependency between classes

Definition

There is a direct cyclical compile dependency between two non-nested, unrelated classes or interfaces. In the case of C++, a linker dependency is also considered

Motivation

Classes that depend on each other can only be understood, maintained, used and reused together. Any change in one of the classes may require changes in the other.

Languages

C++, Java, C#, Delphi

2.5. Cyclical dependency between packages

Definition

Two packages have compile dependencies on each other.

Motivation

Packages are usually employed in order to separate subsystems of a system. Packages that depend on each other can only be understood, maintained, used and reused together. Any change in one of the packages may require changes in the other.

Languages

C++, Java, C#, Delphi

2.6. Cyclical dependency between source files

Definition

There is a cyclical compile dependency between two source files.

Motivation

Source files that depend on each other can only be understood, maintained, used and reused together. Any change in one of the files may require changes in the other.

Languages

C++, Java, C#

2.7. Cyclical method calls

Definition

Two methods call each other directly in a non-polymorphic fashion.

Motivation

Methods that call each other can only be understood, maintained, used and reused together. Any change in one of the methods may require changes in the other.

Languages

C++, Java, C#, Delphi

2.8. Dead attribute

Definition

A private attribute or constant is not referenced anywhere within the declaring class.

Motivation

Attributes that are private to a class but not used by it are redundant and unnecessarily clutter the source code.

Languages

C++, Java, C#, Delphi

2.9. Dead code

Definition

Statements that are outside of any possible control flow path and therefore cannot be executed under any circumstances are considered dead code.

Motivation

Code that is guaranteed to be out of any possible execution path is redundant and clutters method definitions, potentially making them harder to understand. Typical examples that can be reliably detected with automated tools are statements that follow a return or an exception throw.

Languages

C++, Java, C#, Delphi

2.10. Dead imports

Definition

We have a case of dead import when a type is explicitly imported but not used. In other words, there are no compile dependencies to the type, apart from the import.

Motivation

Having very sharp imports/includes (no redundant or general imports) helps in quickly assessing the semantic environment of a class. In the case of C++, unused includes induce unwanted dependencies that can provoke unnecessary recompiles.

Languages

C++, Java, C#

2.11. Dead method

Definition

A dead method is a private method of a class, that is not referenced from anywhere within that class.

Motivation

Methods that are private to a class but not used by it are redundant and unnecessarily clutter the source code.

Languages

C++, Java, C#, Delphi

2.12. Duplicated code

Definition

There are at least 40 consecutive lines of source code (including empty lines and comments) that are duplicated in identical form throughout the system.

Motivation

It is often necessary to use the same or similar functionalities in various places throughout the system. This can be achieved by factoring out the wanted functionality and referencing it where it is needed. The general rule should be to have everything only once in the system. If large blocks of code are duplicated, there is a high risk that errors found in one of the places won't be corrected in all clones and this leads to runtime errors.

Languages

C++, Java, C#, Delphi

2.13. General parameter

Definition

The type of a method parameter is cast to a more specialized type (subtype) in the method.

Motivation

The use of a general type in the interface is misleading for the clients. It suggests a certain generality that may not be actually be there. This can lead to problems that only manifest themselves at runtime.

Languages

C++, Java, C#, Delphi

2.14. Generation conflict

Definition

A direct subclass of a superclass overrides more than 50% of the method implementations inherited from the superclass. The new implementations do not use the inherited implementations (i.e. by calling them).

Motivation

This problem suggests an abuse of the inheritance mechanism in order to reuse a small portion of the functionality.

Languages

C++, Java, C#, Delphi

2.15. God class, attribute form

Definition

A class that declares/defines more than 50 attributes (not constants). The visibility is not relevant.

Motivation

A class that defines a lot of attributes may have trouble guaranteeing the consistency and correctness of its internal state. It is also hard to understand and modify.

Languages

C++, Java, C#, Delphi

2.16. God class, method form

Definition

A class/interface that contain more than 50 methods having at least the visibility of the class/interface itself is considered a god class.

Motivation

Classes which exhibit this problem are hard to understand, use, reuse and maintain, because they have extremely wide interfaces and large size. Often, they are also not cohesive, but rather a sort of container for all kinds of semantically unrelated behavior (e.g. such as a class that offers library functions).

Languages

C++, Java, C#, Delphi

2.17. God file

Definition

A file that contains more than 2000 LOC is considered a god file.

Motivation

Very large files are hard to maintain.

Languages

C++, Java, C#, Delphi

2.18. God method

Definition

A god method is a method whose implementation has more than 200 lines of code.

Motivation

Readability and understandability have a great influence on the maintainability of code. A method having several hundred LOC is neither of the two. This problem differs from “complex method” in the sense that the focus is on size, rather than complexity. A method can be complex but need not be also long.

Languages

C++, Java, C#, Delphi

2.19. God package

Definition

A god package is a package containing more than 50 public non-nested classes or interfaces.

Motivation

Packages and directories are a means of expressing abstractions on a higher level than that of classes and interfaces. If such a package contains too many classes and interfaces, it becomes hard to understand and consequently to maintain.

Languages

C++, Java, C#, Delphi

2.20. Ignored abstraction

Definition

An interface or a class declares a public method, but the interface/class is not used from outside of the inheritance hierarchy.

Motivation

An unused public interface is an unnecessary open window on the subsystem in which it resides. This can result for example in situations when somebody defines an interface for all the classes that he writes.

Languages

C++, Java, C#, Delphi

2.21. Import chaos

Definition

We have a case of import chaos when the imports/includes are either redundant or implicit. In other words, in the following cases:

- a type from the same package is imported
- a type from `java.lang` is explicitly imported
- multiple import/inclusion of the same type
- on-demand imports of entire packages.

Motivation

Having very sharp imports/includes (no redundant or general imports) helps in quickly assessing the semantic environment of a class. In the case of C++, unused includes induce unwanted dependencies that can provoke unnecessary recompiles.

Languages

C++, Java, C#

2.22. Improper name length

Definition

The length of an artifact's name is either shorter as 2 or longer as 50 characters. The following artifacts are considered:

- packages (only the unqualified name)
- files
- classes/interfaces
- methods (does not include parameter list and return type)
- attributes/constants

Motivation

Well chosen identifiers contribute to the legibility of the code and therefore increase its understandability. If the length is too short (e.g. 1, 2 characters) it is rarely expressive enough to capture the necessary semantics. On the other hand, if it's too long, it is hard to read, interpret and remember. Both situations should be avoided in order to have a maintainable system.

Languages

C++, Java, C#, Delphi

2.23. Inappropriate commenting

Definition

A system is considered to be badly commented, if the number of comment lines is either too low or too high. The heuristic measures the absolute difference between the number of code lines (pure code, no comments) and the number of comment lines, and compares the obtained ratio to average values established based on many real life systems. Comments at the end of code lines are not considered.

Motivation

While code commenting can be a double edged sword (in that a bad comment does more harm than no comment), it is generally accepted that commenting increases the understandability and therefore the maintainability of source code.

Languages

C++, Java, C#, Delphi

2.24. Inconsistent operations

Definition

Clusters of semantically related methods are incompletely or inconsistently implemented, which can affect their semantics. In Java, these methods are `Object.equals()` and `Object.hashCode()`. In C++ they are `operator==(void)` and `operator==(int)` (including the “-“ variants), copy constructor and the operator=, as well as the requirement to have virtual destructors if there are virtual methods in the class.

Motivation

Certain methods must remain consistent with other methods at all times, because clients may rely on this assumption. Destroying this consistency can result in bugs.

Languages

C++, Java, C#, Delphi

2.25. Informal documentation

Definition

We have a case of informal documentation in a public method declaration, if that declaration makes no use of any documentation comments specific for specialized infrastructures such as Javadoc or DoxyGen. In practice, we look for method declarations that make no use of `/**`.

Motivation

Documenting the design and source code of a system is a good practice that often eases the job of maintainers. One problem which can arise is that the documentation gets out of sync with the code. The probability of such a mishap decreases significantly, if documentation is generated from the source code itself, because it is generally easier to update the relevant documentation right after a change in the code itself. Mechanisms such as Javadoc or DoxyGen offer a solid infrastructure that supports keeping the documentation and the code close to one another.

Languages

C++, Java, C#

2.26. Interface-Bypass

Definition

A public method is declared in an abstraction (interface or abstract class), which is not used directly by clients. Instead, clients use only direct or indirect descendants of the abstraction.

Motivation

The use of concrete specializations instead of the corresponding abstraction increases the number of dependencies between subsystems. Flexibility and reuse are also negatively affected because the client relies on a concrete type which cannot be replaced with one of its potential siblings.

Languages

C++, Java, C#, Delphi

2.27. Knows of derived

Definition

A superclass or interface has a compile dependency to one of its direct or indirect descendants.

Motivation

If a class depends on its subtypes, it may need to be changed as soon as a change occurs in one of the known subtypes. This is an artificial situation that is against the principles of object oriented programming. Typical scenarios of this problem include methods that check the identity of the hosting object (this) using runtime type identification (instanceof) to vary behavior.

Languages

C++, Java, C#, Delphi

2.28. Late abstraction

Definition

An abstract class has at least one direct or indirect non-abstract superclass.

Motivation

In general, abstract classes are used to express high-level abstractions that capture abstract domain concepts that are widely used and/or specialized throughout the design. Abstract classes that inherit from non-abstract classes are considered bad style, because they violate this assumption and decrease the understandability of the design. Furthermore, the Liskov Substitution Principle is also violated, because the subclass does not possess all of the characteristics of its parent, namely, it cannot instantiate objects.

Languages

C++, Java, C#

2.29. Long parameter list

Definition

A method or constructor has more than 7 call parameters. optional parameters are not counted. Variable parameter lists are counted as single parameters. Methods that are defined in external libraries and are overridden or implemented in the system are not considered.

Motivation

A method or constructor that has lots of parameters is difficult to understand, use and reuse. Typically parameters can be grouped into more complex structures or even objects, which then can be handled more easily. Alternatively, this may also be a sign that the method does too much (see god method, complex method) and it should be split.

Languages

C++, Java, C#, Delphi

2.30. Mini-classes

Definition

A public, non-nested class defines less than 3 methods and less than 3 attributes (including constants) of its own.

Motivation

A very small class may not have a good enough justification to exist on its own. A system with many small classes is harder to understand.

Languages

C++, Java, C#, Delphi

2.31. Mini-packages

Definition

A non-empty package contains less than 3 classes and interfaces.

Motivation

The need for a very small package is questionable since packages are meant as a mechanism to organize classes. Its existence may unnecessarily complicate the architecture of a system.

Languages

C++, Java, C#

2.32. Misleading naming of source files

Definition

The name of a source file is misleading when it does not correspond to any of the class names defined in that file, that are non-nested and have a maximal visibility of all classes in the file. The names must coincide, including their capitalization. In the case of C++, header files are also considered.

Motivation

A filename that is completely out of sync with what's defined inside make finding classes harder.

Languages

C++, Java, C#, Delphi

2.33. Object placebo, attribute form

Definition

A static attribute (including constant) is referenced through an object instance, not the class in which it is defined.

Motivation

Though allowed in all OO languages, accessing static attributes through an object reference is misleading because this may result in extra copies of the existing objects created in order to access the attributes.

Languages

C++, Java, C#, Delphi

2.34. Object placebo, method form

Definition

A static method is referenced through an object instance, not the class that defined it.

Motivation

Though allowed in all OO languages, calling static methods through an object reference is misleading because this may result in extra copies of the existing objects created in order to call the methods.

Languages

C++, Java, C#, Delphi

2.35. Orphan sibling attributes

Definition

A non-static attribute is defined in more than 50% or at least 10 of the direct subclasses (at least 2 in any case), without being defined in any of the subclasses' ancestors.

Motivation

Identically named attributes defined in sibling classes may represent a commonality between the siblings. The right place to define features common to several abstractions is in a supertype of those abstractions. An additional advantage is that the commonality is expressed in one place, and must be understood and tested once.

Languages

C++, Java, C#, Delphi

2.36. Orphan sibling methods

Definition

A method is declared/defined in more than 50% or at least 10 of the direct subclasses (at least 2 in any case), without being defined in any of the subclasses' ancestors.

Motivation

Identically named methods defined in sibling classes may represent a commonality between the siblings. The right place to define features common to several abstractions is in a supertype of those abstractions. An additional advantage is that the commonality is expressed in one place, and must be understood and tested once.

Languages

C++, Java, C#, Delphi

2.37. Overloaded file

Definition

Is a source code file that contains the declaration/definition of more than one non-nested class.

Motivation

While generally allowed by the considered programming languages, defining several non-nested classes in a single file makes them harder to find and may induce unwanted compile dependencies, making the reuse of a single class in a different context harder.

Languages

C++, Java, C#, Delphi

2.38. Permissive visibility, attribute form

Definition

The declared visibility of an attribute (no constants, no package-private visibility) is more permissive than actually needed. In other words, it is possible to reduce the visibility without compromising the ability to compile and link the system.

Motivation

Declaring attributes more permissive than needed unnecessarily exposes the internals of a class to the outside and violates information hiding rules.

Languages

C++, Java, C#, Delphi

2.39. Permissive visibility, method

Definition

A protected method of a class is not used or overridden in any of its descendants, and itself does not override/implement any method inherited from a supertype.

Motivation

Declaring methods more permissive than needed unnecessarily exposes the internals of a class to the outside and complicates its exposed interface.

Languages

C++, Java, C#, Delphi

2.40. Polymorphic calls in constructor

Definition

In an object constructor there are direct polymorphic calls to methods (including inherited ones) of the class. The fact that the called method is overridden or not in a descendant is not considered.

Motivation

Calling virtual methods from within a constructor may result in a call to an override method from a subclass which may then operate on an incompletely initialized object instance. This situation may lead to bugs which are very hard to track down.

Languages

C++, Java, Delphi

2.41. Polymorphism placebo

Definition

A static method in a superclass, while visible in a subclass, is overridden in the subclass.

Motivation

Structurally, this situation resembles that of a polymorphic method hierarchy and may be easily mistaken with it. In combination with the above described Object placebo problems, this problem may lead to code that is difficult to understand.

Languages

C++, Java, C#

2.42. Refused bequest, implementation form

Definition

A method's implementation belonging to a superclass is overridden in the majority of its subclasses, with non-empty implementations, without using the inherited implementation (i.e. calling it).

Motivation

In general, it is allowed for a subclass to override inherited methods with their own specialized versions. If however this happens in the majority of the subclasses, it could mean that the inherited implementation of the method (the method's body) is not general enough to justify its place in the superclass. A further sign that confirms this situation is when the overridden methods do not call the superclass version. In such situations, the inheritance hierarchy is not optimal.

Languages

C++, Java, C#, Delphi

2.43. Refused bequest, interface form

Definition

A subclass doesn't want to support the entire interface inherited from its parent. In other words, the subclass either:

- overrides/implements an inherited method/signature with an empty or trivial body (i.e. "return null" or "throw new Exception...") or,
- reduces the visibility of an inherited method/signature by using private inheritance (only C++).

Motivation

In general, all inheritance relations should obey to the so called Liskov substitution principle (LSP). This insures that objects of the subclass can always be used instead of objects of the superclass in a safe fashion. One of the fundamental requirements of LSP is that subclasses fully support all inherited interfaces. This rule has a semantic nature, but there are some structural signs that indicate the possibility of a violation of LSP. These are cases, in which a class employs private inheritance, or cases in which the subclass overrides the unwanted interface's method with empty methods, or methods that just throw exceptions.

Using objects of such classes in place of objects of the superclass can cause runtime errors to occur.

Languages

C++, Java

2.44. Reversed package and inheritance hierarchies

Definition

A class that is contained in a package inherits directly from a type that is contained in a sub-package of the first package.

Motivation

The hierarchical structure of packages suggests a process of top-down refinement. A sub-package refines a particular aspect of the package containing it. As such an inheritance (specialization) of a class in a sub-package by a class in the enclosing package is unexpected.

Languages

C++, Java, C#

2.45. Risky code

Definition

We have an instance of risky code when typical techniques of defensive programming are not being used. Examples of such code include empty exception handlers, missing breaks or default branch in a switch statement.

Motivation

Failing to handle every possible case / exception may lead to unexpected behavior.

Languages

C++, Java, C#, Delphi

2.46. Similar unrelated abstractions

Definition

There are at least two, public, non-nested, unrelated classes which define at least 10 identical, non-inherited method signatures. Constructors and destructors are not counted.

Motivation

Duplication at the public interface level is usually a sign of badly conceived class hierarchies. If several classes within the same hierarchy share a number of public operations, it may be the case that a common abstraction capturing this set of shared methods is missing.

Languages

C++, Java, C#, Delphi

2.47. Simulated polymorphism

Definition

An object is checked several times in order to determine its runtime type within a method.

Motivation

Checking for runtime type information and performing different actions based on this type is considered bad OO style and makes maintenance more difficult. It is a common mistake made by programmers used to develop code in procedural languages and typically requires maintaining large switch statements checking for the type of a variable. The problem with these large switches is that they have to be updated every time a new class is added to the type hierarchy. OO languages offer a more elegant solution to cope with this situation, namely polymorphism.

Languages

C++, Java, C#, Delphi

2.48. Type name duplication

Definition

Two or more public classes and/or interfaces in the system have the same name. Case differences are ignored.

Motivation

Classes and interfaces are usually not referred to in the code using the fully qualified name. Having several identically named classes in the system may lead to misunderstandings when reading code fragments that reference such classes.

Languages

C++, Java, C#, Delphi

2.49. Unfinished code

Definition

Is detected based on comments in the code, which suggest that the respective code fragment is probable to change. Such comments include the words such as “todo”, “hack” or “fixme”.

Motivation

The mentioned tags usually mark code fragments that are very likely to change in the near future. Having too many such fragments makes the job harder for maintainers and developers alike.

Languages

C++, Java, C#, Delphi

2.50. Variables having constant value

Definition

A class attribute (static) is not declared as a constant (final or const). However, it is only written to once, upon initialization.

Motivation

Class attributes that are meant as constants should be declared as final or const. This helps eliminate the semantic ambiguity inherent in a constant defined using an ordinary attribute definition.

Languages

C++, Java, C#, Delphi

2.51. Violation of data encapsulation

Definition

A non-inner class defines public attributes that are not constants (e.g. static final), and there is at least one read or write access to such an attribute from a different class that is not an inner class of the first.

Motivation

One of the fundamental goals of object orientation is the encapsulation of data and behavior operating on that data. Ideally, the only visible part to the outside of a class should be a set of higher-level services that use but do not expose the internal details of the class' implementation. Thus, the class is able to manage and guarantee a correct and consistent internal state. A direct manipulation of the internal state from outside of the object could lead to runtime errors and should therefore not be allowed.

Languages

C++, Java, C#, Delphi

2.52. Violation of naming convention

Definition

This rule checks for consistency in the naming conventions used for entities throughout the system. The rules given below serve as an example only and can be adapted for the system at hand.

- A package name is described by the regular expression `\[a-zA-Z]\[a-zA-Z0-9_]*`
- A class/interface name is described by the regular expression `\[A-Z]\[a-zA-Z0-9_]*`
- A file name is described by the regular expression `\[a-zA-Z]\[a-zA-Z0-9_]*\|\|\|\.[a-zA-Z]+`
- A method name in JAVA (except constructors) is described by the regular expression `\[a-z]\[a-zA-Z0-9_]*`
- A method name in C++ is described by the regular expression `\[A-Z]\[a-zA-Z0-9_]*`
- A constant name is described by the regular expression `\[A-Z]\[A-Z0-9_]*`
- An attribute identifier is described by the regular expression `\[a-z]\[a-zA-Z0-9_]*`

Motivation

Naming conventions are an important part of team development. They build up a common set of expectations on the implementation artifacts. If code conventions are followed, system understanding becomes easier and maintenance costs decrease.

Languages

C++, Java, C#, Delphi