OBJECT-ORIENTED METRICS pdf

1: OOAD Testing and Quality Assurance

Lines of code and functional point metrics can be used for estimating object-oriented software projects. However, these metrics are not appropriate in the case of incremental software development as they do not provide adequate details for effort and schedule estimation.

Object Oriented Metrics Object-Oriented Analysis and Design of software provide many benefits such as reusability, decomposition of problem into easily understood object and the aiding of future modifications. But the OOAD software development life cycle is not easier than the typical procedural approach. Therefore, it is necessary to provide dependable guidelines that one may follow to help ensure good OO programming practices and write reliable code. Object-Oriented programming metrics is an ascpect to be considered. Metrics to be a set of standards against which one can measure the effectiveness of Object-Oriented Analysis techniques in the design of a system. OO metrics which can be applied to analyze source code as an indicator of quality attributes. The source code could be any OO language. The higher the degree of uncoupled object the more objects will suitable for reuse within the same applications and within other applications. Testability is likely to degrade with a more highly coupled system of objects. Object interaction complexity associated with coupling can lead to increased error generation during development. Low cohesion adds complexity which can translate into a reduction in application reliability. Objects which are less dependent on other objects for data are likely to be more reusable. The smaller the number of implementation locations for the average task, the less likely that errors were made during coding The more highly factored an inheritance hierarchy is the greater degree to which method reuse occurs The more highly factored an apllication is, the smaller the number of implementation locations for the average method Degree of Reuse of Inheritance Methods Percent of Potential Method Uses Actually Reused PP: Greater method complexity is likely to lead to a lower degree of overall application comprehensibility. Greater method complexity is likely to adversely affect application reliability. More complex methods are likely to be more difficult to test. An application constructed with more finely granular objects i. More finely granular objects should also be more reusable. They have defined six metrics for the OO design. The number of methods and the complexity of methods involved is a predictor of how much time and effort is required to develop and maintain the class. The larger the number of methods in a class the greater the potential impact on children, since children will inherit all the methods defined in the class. Classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse. The deeper a class is in the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behavior. Deeper trees constitute greater design complexity, since more methods and clases are involved. The deeper a particular class is in the hierarchy, the greater the potential reuse of inherited methods. The greater the number of children, the greater the reuse, since inheritance is a form of reuse. The greater the number of children, the greater the likelihood of improper abstraction of the parent class. If a class has a large number of children, it may be a case of misuse of subclassing. The number of children gives an idea of the potential influence a class has on the design. If a class has a large number of children, it may require more testing of the methods in that class. Coupling between object classes CBO It is defined as the count of the classes to which this class is coupled. Coupling is defined as: Two classes are coupled when methods declared in one class use methods or instance variables of the other class. The more independent a class is, the easier it is to reuse it in another application. In order to improve modularity and promote encapsulation, inter-object class couples should be kept to a minimum. The larger the number of couples, the higher the sensitivity to changes in other parts of the design, and therefore maintenance is more difficult. A measure of coupling is useful to determine how complex the testing of various parts of a design are likely to be. The higher the inter-object class coupling, the more rigorous the testing needs to be. Response For a Class RFC It is defined as number of methods in the set of all methods that can be invoked in response to a message sent to an object of a class. If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes more complicated since it requires a greater level of understanding on the part of the tester. The larger the number of methods that can be

invoked from a class, the greater the complexity of the class. A worst case value for possible responses will assist in appropriate allocation of testing time. Lack of Cohesion in Methods LCOM It is defined as the number of different methods within a class that reference a given instance variable. Cohesiveness of methods within a class is desirable, since it promotes encapsulation. Lack of cohesion implies classes should probably be split into two or more subclasses. Any measure of disparateness of methods helps identify flaws in the design of classes. Low cohesion increases complexity, thereby increasing the likelihood of errors during the development process. Drawbacks Measuring comlexity of a class is subject to bias They cannot give a good size and effort estimation of software These metrics seem only to bring the design phase into play, and does not provide adequate coverage in terms of planning.

OBJECT-ORIENTED METRICS pdf

2: Project Metrics Help - Chidamber & Kemerer object-oriented metrics suite

Metrics are paramount in every engineering discipline. Software engineering, however, is not considered a classical engineering activity for several reasons. In general, if a software system is seen to deliver the required functionality, only few people if any care about the internals. Moreover.

This site contains older material on Eiffel. For the main Eiffel page, see http: The request I hear most commonly, when talking to project managers using object technology or about to use it, is for more measurement tools. Some of those people would kill for anything that can give them some kind of quantitative grasp on the software development process. There is in fact an extensive literature on software metrics, including for object-oriented development, but surprisingly few publications are of direct use to actual projects. Metrics are not everything. This also puts in perspective some of the comments published recently in this magazine July by Walter Tichy and Marvin Zelkowitz on the need for more experimentation, which were largely a plea for more quantitative data. I agree with their central argument -- that we need to submit our hypotheses to the test of experience; but when Tichy writes Zelkowitz and Wallace also surveyed journals in physics, psychology, and anthropology and again found much smaller percentages of unvalidated papers [i. In an engineering discipline we cannot tolerate the fuzziness that is probably inevitable in social sciences. If we are looking for rigor, the tools of mathematical logic and formal reasoning are crucial, even though they are not quantitative. Still, we need better quantitative tools. Numbers help us understand and control the engineering process. In this column I will present a classification of software metrics and five basic rules for their application. Types of metrics The first rule of quantitative software evaluation is that if we collect or compute numbers we must have a specific intent related to understanding, controlling or improving software and its production. This implies that there are two broad kinds of metrics: Product metrics include two categories: External product metrics include: Product non-reliability metrics, assessing the number of remaining defects. Functionality metrics, assessing how much useful functionality the product provides. Cost metrics, assessing the cost of purchasing and using a product. Internal product metrics include: Size metrics, providing measures of how big a product is internally. Complexity metrics closely related to size , assessing how complex a product is. Style metrics, assessing adherence to writing guidelines for product components programs and documents. Cost metrics, measuring the cost of a project, or of some project activities for example original development, maintenance, documentation. Effort metrics a subcategory of cost metrics , estimating the human part of the cost and typically measured in person-days or person-months. Advancement metrics, estimating the degree of completion of a product under construction. Process non-reliability metrics, assessing the number of defects uncovered so far. Reuse metrics, assessing how much of a development benefited from earlier developments. Internal and external metrics The second rule is that internal and product metrics should be designed to mirror relevant external metrics as closely as possible. Clearly, the only metrics of interest in the long run are external metrics, which assess the result of our work as perceptible by our market. Internal metrics and product metrics help us improve this product and the process of producing it. They should always be designed so as to be eventually relevant to external metrics. Object technology is particularly useful here because of its seamlessness properties, which reduces the gap between problem structure and program structure the "Direct Mapping" property. In particular, one may argue that in an object-oriented context the notion of function point, a widely accepted measure of functionality, can be replaced by a much more objective measure: Designing metrics The third rule is that any metric applied to a product or project should be justified by a clear theory of what property the metric is intended to help estimate. The set of things we can measure is infinite, and most of them are not interesting. For example I can write a tool to compute the sum of all ASCII character codes in any program, modulo 53, but this is unlikely to yield anything of interest to product developers, product users, or project managers. A simple example is a set of measurements that we performed some time ago on the public-domain EiffelBase library of fundamental data structures and algorithms, reported in the book Reusable Software Prentice Hall. One of the things we counted was the number of arguments to a feature attribute or routine over classes and features, and found an average

of 0. We were not measuring this particular property in the blind: These figures show a huge decrease as compared to the average number of arguments for typical non-O-O subroutine libraries, often 5 or more, sometimes as much as Note that a C or Fortran subroutine has one more argument than the corresponding O-O feature. Quantitative arguments such as provided by the EiffelBase measurements provide some concrete evidence to back the O-O claims. The second rule requires a theory, and implies that the measurements will only be as good as the theory. Indeed, the correlation between small number of feature arguments and ease of library use is only a hypothesis. Authors such as Zelkowitz and Tichy might argue that the hypothesis must be validated through experimental correlation with measures of ease of use. They would have a point, but the first step is to have a theory and make it explicit. Experimental validation is seldom easy anyway, given the setup of many experiments, which often use students under the sometimes dubious assumption that their reactions can be used to predict the behavior of professional programmers. In addition, it is very hard to control all the variables. For example I recently found out, by going back to the source, that a nineteen-seventies study often used to support the use of semicolons as terminators rather than separators seemed to rely on an unrealistic assumption which casts doubt on the results. Starting from several hundred informal methodological rules in the book "Reusable Software" and others, they identified the elements of these rules that could be subject to quantitative evaluation, defined the corresponding metrics, and produced tools that evaluate these metrics on submitted software. Project managers or developers using these tools can assess the values of these measurements on their products. In particular, you can compare the resulting values to industry-wide standards or to averages measured over your previous projects. This brings the fourth rule, which states that measurements are usually most useful in relative terms. Calibrating metrics More precisely, the fourth rule is that most measurements are only meaningful after calibration and comparison to earlier results. This is particularly true of cost and reliability metrics. As you move on to new projects, you can use the model with more and more confidence based on comparisons with other projects. Similarly, many internal product metrics are particularly useful when taken relatively. Presented with an average argument count measure of 4 for your newest library, you will not necessarily know what it means -- good, bad, irrelevant? Assessed against published measures of goodness, or against measures for previous projects in your team, it will become more meaningful. Particularly significant are outlying points: This is where tools such as the Monash suite can be particularly useful. Metrics and the measuring process The fifth rule is that the benefits of a metrics program lie in the measuring process as well as in its results. The software metrics literature often describes complex models purporting to help predict various properties of software products and processes by measuring other properties. It also contains lots of controversy about the value of the models and their predictions. The very process of collecting these measurements leads as long as we confine ourselves to measurements that are meaningful, at least by some informal criteria to a better organization of the software process and a better understanding of what we are doing. To quote Emmanuel Girard, a software metrics expert, in his advice for software managers:

3: oop - Metrics & Object-oriented programming - Stack Overflow

"This well-written book is an important piece of work that takes the seemingly forgotten art of object-oriented metrics to the next level in terms of relevance and usefulness." Richard C. Gronback, Chief Scientist, Borland Software Corporation.

A high WMC has been found to lead to more faults. Classes with many methods are likely to be more more application specific, limiting the possibility of reuse. WMC is a predictor of how much time and effort is required to develop and maintain the class. A large number of methods also means a greater potential impact on derived classes, since the derived classes inherit some of the methods of the base class. Search for high WMC values to spot classes that could be restructured into several smaller classes. What is a good WMC? Different limits have been defined. One way is to limit the number of methods in a class to, say, 20 or This allows large classes but most classes should be small. Guidelines for Improving Quality. Constructors and event handlers are also counted as methods. WMC includes only those methods that are defined in the class, not any inherited methods. Overriding and shadowing methods defined in the class are counted, since they form a new implementation of a method. The reasoning behind this is that each of Get, Set and Let provides a separate way to access to the underlying property and is essentially a method of the class. The alternative way to using properties would be to write accessor functions: Function getProperty and Sub setProperty. Both of these would also be counted in WMC the same way we count each property accessor. In principle, we could weigh each method by its size or complexity. This is not implemented, though. A Metrics suite for Object Oriented design. Sloan School of Management E Deep trees as such indicate greater design complexity. Inheritance is a tool to manage complexity, really, not to not increase it. As a positive factor, deep trees promote reuse because of method inheritance. A high DIT has been found to increase faults. According to them, root and deepest classes are consulted often, and due to familiarity, they have low fault-proneness compared to classes in the middle. A recommended DIT is 5 or less. Some sources allow up to 8. It sounds safe to assume that a deep inheritance tree is detrimental in VB as well. Project Analyzer takes only implementation inheritance Inherits statement, not Implements statement into account when calculating DIT. When a class inherits directly from System. This is because the unknown class eventually inherits from System. Object and 2 is the minimum inheritance depth. It could also be more. The prediction of faulty classes using object-oriented design metrics. The Journal of Systems and Software 56 NET one uses the Inherits statement to derive sub-classes. Depth is generally better than breadth, since it promotes reuse of methods through inheritance. Inheritance levels can be added to increase the depth and reduce the breadth. A high NOC, a large number of child classes, can indicate several things: High reuse of base class. Inheritance is a form of reuse. Base class may require more testing. Improper abstraction of the parent class. In such a case, it may be necessary to group related classes and introduce another level of inheritance. High NOC has been found to indicate fewer faults. This may be due to high reuse, which is desirable. The class is potentially influencing a large number of descendant classes. This can be a sign of poor design. A redesign may be required. Not all classes should have the same number of sub-classes. Classes higher up in the hierarchy should have more sub-classes then those lower down. The uses relationship can go either way: Multiple accesses to the same class are counted as one access. Only method calls and variable references are counted. Other types of reference, such as use of constants, calls to API declares, handling of events, use of user-defined types, and object instantiations are ignored. If a method call is polymorphic either because of Overrides or Overloads , all the classes to which the call can go are included in the coupled count. High CBO is undesirable. Excessive coupling between object classes is detrimental to modular design and prevents reuse. The more independent a class is, the easier it is to reuse it in another application. In order to improve modularity and promote encapsulation, inter-object class couples should be kept to a minimum. The larger the number of couples, the higher the sensitivity to changes in other parts of the design, and therefore maintenance is more difficult. A high coupling has been found to indicate fault-proneness. Rigorous testing is thus needed. For example a system in which a single class has very high fan-out and all other classes have low or zero

fan-outs, we really have a structured, not an object oriented, system. The definition of CBO deals with the instance variables and all the methods of a class. Thus, Shared variables class variables are not taken into account. On the contrary, all method calls are taken into account, whether Shared or not. This distinction does not seem to make any sense, but we follow the original definition. If a call is polymorphic in that it is to an Interface method in. NET, this is not taken as a coupling to either the Interface or the classes that implement the interface. This is a limitation of the implementation, not the definition of CBO. In this implementation of CBO, when a child class calls its own inherited methods, it is coupled to the parent class where the methods are defined. The original CBO definition does not define if inheritance should be treated in any specific way. Therefore, we follow the definition and treat inheritance as if it was regular coupling. San Diego State University. Fraunhofer Institute for Experimental Software Engineering. Sahraoui, Robert Godin, Thierry Miceli: RFC is simply the number of methods in the set. Since RFC specifically includes methods called from outside the class, it is also a measure of the potential communication between the class and other classes. A large RFC has been found to indicate more faults. Classes with a high RFC are more complex and harder to understand. Testing and debugging is complicated. A worst case value for possible responses will assist in appropriate allocation of testing time. RFC is the original definition of the measure. It counts only the first level of calls outside of the class. RFC was originally defined as a first-level metric because it was not practical to consider the full call tree in manual calculation. It regards all subs, functions, properties and API declares as methods, whether in classes or other modules. Calls to property Set, Let and Get are all counted separately. Calls to VB library functions, such as print, are not counted. Calls are counted to declared API procedures. If COM libraries were included in the analysis, calls to procedures in those libraries are also counted. Dead methods and their callees are included in the measure. This metric has received a great deal of critique and several alternatives have been developed. LCOM1 is described among the other cohesion metrics. The study analyzed 3 systems and classified their quality.

OBJECT-ORIENTED METRICS pdf

4: Software package metrics - Wikipedia

This book provides a number of specific metrics that apply to object-oriented software projects. The metrics are based on measurements and derived advice drawn from a number of actual projects that have successfully used object technology to deliver products.

Software Engineering Lines of code and functional point metrics can be used for estimating object-oriented software projects. However, these metrics are not appropriate in the case of incremental software development as they do not provide adequate details for effort and schedule estimation. Thus, for object-oriented projects, different sets of metrics have been proposed. These are listed below. Number of scenario scripts: Scenario scripts are a sequence of steps, which depict the interaction between the user and the application. A number of scenarios is directly related to application size and number of test cases that are developed to test the software, once it is developed. Note that scenario scripts are analogous to use-cases. Number of key classes: Key classes are independent components, which are defined in object -oriented analysis. Number of support classes: Classes, which are required to implement the system but are indirectly related to the problem domain, are known as support classes. For example, user interface classes and computation class are support classes. It is possible to develop a support class for each key class. Average number of support classes per key class: Key classes are defined early in the software project while support classes are defined throughout the project. The estimation process is simplified if the average number of support classes per key class is already known. A collection of classes that supports a function visible to the user is known as a subsystem. Identifying subsystems makes it easier to prepare a reasonable schedule in which work on subsystems is divided among project members. The afore-mentioned metrics are collected along with other project metrics like effort used, errors and defects detected, and so on. After an organization completes a number of projects, a database is developed, which shows the relationship between object-oriented measure and project measure. This relationship provides metrics that help in project estimation. Dinesh authors the hugely popular Computer Notes blog. Where he writes how-to guides around Computer fundamental , computer software, Computer programming, and web apps. For any type of query or something that you think is missing, please feel free to Contact us.

OBJECT-ORIENTED METRICS pdf

5: Applying and Interpreting Object-oreinted Metrics

Object-Oriented programming metrics is an ascpect to be considered. Metrics to be a set of standards against which one can measure the effectiveness of Object-Oriented Analysis techniques in the design of a system.

As defined, it is a number of classes in a given package, which depends on the classes in other packages. It enables us to measure the vulnerability of the package to changes in packages on which it depends. Preferred values for the metric Ce are in the range of 0 to 20, higher values cause problems with care and development of code. Afferent Coupling Ca This metric is an addition to metric Ce and is used to measure another type of dependencies between packages, i. It enables us to measure the sensitivity of remaining packages to changes in the analysed package. High values of metric Ca usually suggest high component stability. This is due to the fact that the class depends on many other classes. Preferred values for the metric Ca are in the range of 0 to Instability I This metric is used to measure the relative susceptibility of class to changes. According to the definition instability is the ration of outgoing dependencies to all package dependencies and it accepts value from 0 to 1. The metric is defined according to the formula: Ce â€" outgoing dependencies, Ca â€" incoming dependencies Pic. On the basis of value of metric I we can distinguish two types of components: The ones having many outgoing dependencies and not many of incoming ones value I is close to 1 , which are rather unstable due to the possibility of easy changes to these packages; The ones having many incoming dependencies and not many of outgoing ones value I is close to 0 , therefore they are rather more difficult in modifying due to their greater responsibility. Preferred values for the metric I should fall within the ranges of 0 to 0. Packages should be very stable or unstable, therefore we should avoid packages of intermediate stability. Abstractness A This metric is used to measure the degree of abstraction of the package and is somewhat similar to the instability. Regarding the definition, abstractness is the number of abstract classes in the package to the number of all classes. Tabstract â€" number of abstract classes in a package, Tconcrete â€" number of concrete classes in a package Preferred values for the metric A should take extreme values close to 0 or 1. Packages that are stable metric I close to 0 , which means they are dependent at a very low level on other packages, should also be abstract metric A close to 1. In turn, the very unstable packages metric I close to 1 should consist of concrete classes A metric close to 0. Additionally, it is worth mentioning that combining abstractness and stability enabled Martin to formulate thesis about the existence of main sequence Pic. Classes that were well designed should group themselves around this graph end points along the main sequence. Normalized Distance from Main Sequence D This metric is used to measure the balance between stability and abstractness and is calculated using the following formula: A- abstractness, I - instability The value of metric D may be interpreted in the following way; if we put a given class on a graph of the main sequence Pic. In addition, the two extremely unfavourable cases are considered: All metrics discussed in this article can be measured for projects in. NET using NDepend tool.

OBJECT-ORIENTED METRICS pdf

6: Henderson-Sellers, Object-Oriented Metrics: Measures of Complexity | Pearson

2 OO Metrics: Introduction Measurement and metrics are key components of any engineering discipline. The use of metrics for OO systems has progresses much.

Next Page Once a program code is written, it must be tested to detect and subsequently handle all errors in it. A number of schemes are used for testing purposes. Another important aspect is the fitness of purpose of a program that ascertains whether the program serves the purpose which it aims for. The fitness defines the software quality. Testing Object-Oriented Systems Testing is a continuous activity during software development. In object-oriented systems, testing encompasses three levels, namely, unit testing, subsystem testing, and system testing. Unit Testing In unit testing, the individual classes are tested. It is seen whether the class attributes are implemented as per design and whether the methods and the interfaces are error-free. Unit testing is the responsibility of the application engineer who implements the structure. Subsystem Testing This involves testing a particular module or a subsystem and is the responsibility of the subsystem lead. It involves testing the associations within the subsystem as well as the interaction of the subsystem with the outside. Subsystem tests can be used as regression tests for each newly released version of the subsystem. System Testing System testing involves testing the system as a whole and is the responsibility of the quality-assurance team. The team often uses system tests as regression tests when assembling new releases. Object-Oriented Testing Techniques Grey Box Testing The different types of test cases that can be designed for testing object-oriented programs are called grey box test cases. Testing starts from the individual classes to the small modules comprising of classes, gradually to larger modules, and finally all the major subsystems. A good quality software does exactly what it is supposed to do and is interpreted in terms of satisfaction of the requirement specification laid down by the user. Quality Assurance Software quality assurance is a methodology that determines the extent to which a software product is fit for use. Object-Oriented Metrics Metrics can be broadly classified into three categories: Project Metrics Project Metrics enable a software project manager to assess the status and performance of an ongoing project.

7: The Role of Object-Oriented Metrics

Software Metrics in the Object-Oriented Paradigm Understanding the object-oriented paradigm is the first step toward defining metrics for that paradigm. The study of the object-oriented paradigm results in object-oriented concepts such as object, class, at- tributes, inheritance, method, and message passing.

Defines the structural properties of classes, unique within a class, generally a noun. Class A set of objects that share a common structure and common behavior manifested by a set of methods; the set serves as a template from which object can be instantiated created. Cohesion The degree to which the methods within a class are related to one another. Inheritance A relationship among classes, wherein an object in a class acquires characteristics from one or more other classes. Instantiation The process of creating an instance of the object and binding or adding the specific data. Message A request that an object makes of another object to perform an operation. Method An operation upon on object, defined as part of the declaration of a class. Object An instantiation of some class which is able to save a state information and which offers a number of operations to examine or affect this state. Operation An action performed by or on an object, available to all instances of class, need not be unique. Key Object Oriented Terms for Metrics The new object oriented development methods have their own terminology to reflect the new structural concepts. Referencing Figure 1, an object oriented system starts by defining a class Class A that contains related or similar attributes and operations some operations are methods. The classes are used as the basis for objects Object A1. A child class inherits all of the attributes and operations from its parent class, in addition to having its own attributes and operations. A child class can also become a parent class for other classes, forming another branch in the hierarchical tree structure. When an object is created to contain data or information, it is an instantiation of the class. Classes interact or communicate by passing messages. When a message is passed between two classes, the classes are coupled. These specific terms are defined in Table 2 [1, 4, 9]. The class of Appliance Departments will have an attribute of Category. Specific named departments are objects. Methods are operations done on an object. Examples of what all store departments need to do are Display merchandise, Give credit, and Exchange merchandise. The Clothing Department will inherit these methods but also have Dressing rooms. Traditional Metrics There are many metrics that are applied to traditional functional development. The SATC, from experience, has identified three of these metrics that are applicable to object oriented development: Complexity, Size, and Readability. To measure the complexity, the cyclomatic complexity is used. It is a count of the number of test cases that are needed to test the method comprehensively. The formula for calculating the cyclomatic complexity is the number of edges minus the number of nodes plus 2. For a sequence where there is only one path, no choices or option, only one test case is needed. An IF loop however, has two choices, if the condition is true, one path is tested; if the condition is false, an alternative path is tested. Figure 3 shows examples of calculations for the cyclomatic complexity for four basic programming structures. Example Calculations Cyclomatic Complexity A method with a low cyclomatic complexity is generally better. This may imply decreased testing and increased understandability or that decisions are deferred through message passing, not that the method is not complex. Cyclomatic complexity cannot be used to measure the complexity of a class because of inheritance, but the cyclomatic complexity of individual methods can be combined with other measures to evaluate the complexity of the class. Although this metric is specifically applicable to the evaluation of Complexity, it also is related to all of the other attributes [3, 5, 6, 7, 11]. Size Size of a class is used to evaluate the ease of understanding of code by developers and maintainers. Size can be measured in a variety of ways. These include counting all physical lines of code, the number of statements, the number of blank lines, and the number of comment lines. Lines of Code LOC counts all lines. Executable Statements EXEC is a count of executable statements regardless of number of physical lines of code. Executable statements is the measure least influenced by programmer or language style. However, since size affects ease of understanding by the developers and maintainers, classes and methods of large size will always pose a higher risk. Comment Percentage The line counts done to compute the Size metric can be expanded to include a count of the number of comments, both on-line with code and stand-alone. The

comment percentage is calculated by the total number of comments divided by the total lines of code less the number of blank lines. Since comments assist developers and maintainers, higher comment percentages increase understandability and maintainability. Object-Oriented Specific Metrics As discussed, many different metrics have been proposed for object oriented systems. The object oriented metrics that were chosen by the SATC measure principle structures that, if improperly designed, negatively affect the design and code quality attributes. The selected object oriented metrics are primarily applied to the concepts of classes, coupling, and inheritance. Preceding each metric, a brief description of the object oriented structure is given. For some of the object-oriented metrics discussed here, multiple definitions are given; researchers and practitioners have not reached a common definition or counting methodology. In some cases, the counting method for a metric is determined by the software analysis package being used to collect the metrics. Recall, a class is a template from which objects can be created. This set of objects shares a common structure and a common behavior manifested by the set of methods. A method is an operation upon an object and is defined in the class declaration. A message is a request that an object makes of another object to perform an operation. The operation executed as a result of receiving a message is called a method. Cohesion is the degree to which methods within a class are related to one another and work together to provide well-bounded behavior. Effective object oriented designs maximize cohesion because cohesion promotes encapsulation. Coupling is a measure of the strength of association established by a connection from one entity to another. Classes objects are coupled when a message is passed between objects; when methods declared in one class use methods or attributes of another class. Inheritance is the hierarchical relationship among classes that enables programmers to reuse previously defined objects including variables and operators. Object Oriented Application Example B. Weighted Methods per Class WMC The WMC is a count of the methods implemented within a class or the sum of the complexities of the methods method complexity is measured by cyclomatic complexity. The second measurement is difficult to implement since not all methods are assessable within the class hierarchy due to inheritance. The number of methods and the complexity of the methods involved is a predictor of how much time and effort is required to develop and maintain the class. The larger the number of methods in a class, the greater the potential impact on children; children inherit all of the methods defined in the parent class. Classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse. Response for a Class RFC The RFC is the count of the set of all methods that can be invoked in response to a message to an object of the class or by some method in the class. This includes all methods accessible within the class hierarchy. This metric looks at the combination of the complexity of a class through the number of methods and the amount of communication with other classes. The larger the number of methods that can be invoked from a class through messages, the greater the complexity of the class. If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes complicated since it requires a greater level of understanding on the part of the tester. A worst case value for possible responses will assist in the appropriate allocation of testing time. A highly cohesive module should stand alone; high cohesion indicates good class subdivision. Lack of cohesion or low cohesion increases complexity, thereby increasing the likelihood of errors during the development process. High cohesion implies simplicity and high reusability. High cohesion indicates good class subdivision. Classes with low cohesion could probably be subdivided into two or more subclasses with increased cohesion. Figure 5 shows a design with high lack of cohesion because there are relatively few common attributes and methods among the objects. Because the objects have few methods in common, there is a high lack of cohesion. This implies further abstraction is needed â€” similar objects need to be grouped together by creating child classes for them. It is measured by counting the number of distinct non-inheritance related class hierarchies on which a class depends. Excessive coupling is detrimental to modular design and prevents reuse. The more independent a class is, the easier it is reuse in another application. The larger the number of couples, the higher the sensitivity to changes in other parts of the design and therefore maintenance is more difficult. Strong coupling complicates a system since a class is harder to understand, change or correct by itself if it is interrelated with other classes. Complexity can be reduced by designing systems with the weakest possible coupling between classes. This improves modularity and promotes encapsulation. Here two departments are identified as Jackets

and Trousers. They both have the same attributes and the same methods. This implies that the design in Figure 6 is probably not the most efficient design and these departments should be combined into one class. Example of Excessive Coupling B. Depth of Inheritance Tree DIT The depth of a class within the inheritance hierarchy is the maximum number of steps from the class node to the root of the tree and is measured by the number of ancestor classes. The deeper a class is within the hierarchy, the greater the number methods it is likely to inherit making it more complex to predict its behavior. Deeper trees constitute greater design complexity, since more methods and classes are involved, but the greater the potential for reuse of inherited methods. The DIT for Clothing is 1. Number of Children NOC The number of children is the number of immediate subclasses subordinate to a class in the hierarchy. It is an indicator of the potential influence a class can have on the design and on the system.

8: Object-Oriented Metrics: LCOM

I need a software to determine (specify) object oriented metrics (WMC,DIT,RFC,NOC,CBO,LCOM,Percent_Pub_Data,Dep_On_Child).Do you know any software to find these metrics from a software. I know.

9: Object Oriented Metrics in Software Engineering

These metrics include three traditional metrics adapted for an object oriented environment, and six "new" metrics to evaluate the principle object oriented structures and concepts. The metrics are first defined, then using a very simplistic object oriented example, the metrics are applied.

OBJECT-ORIENTED METRICS pdf

Running after 40. Just your type full zip Cleveland school survey Copper and its alloys Reels 151-162. Sixth State Militia, Cavalry All About Music Technology in Worship Trials and rewards by F. H. Rayer. Good Night, Gorilla (Mathematics Focus) The Secret Agent (Vintage Classics) The End of Management and the Rise of Organizational Democracy Symbols for communication Beer for beginners. Multimodal Imaging and Hybrid Scanners Black, white, and in color 1967 ford p350 shop manual Kist and tell Matt Barnard. Prayers to Mary for a Happy Death 1186 Economics and Theology Ssc cgl tier 3 question paper Exploring the Christian way During the second five-year plan. Forces of warmachine cygnar mk3 Drama stage and audience Health surveillance by routine procedures. A room called Remember Introduction : a personal note from the Arps The ORVIS beginners guide to birdwatching Handbook of Business Valuation Give the boys a great big hand After-words : what follows post-development? Did the Vikings grow their own food? A dictionary of literary terms cuddon Io programming language tutorial Dancing to the Rocky Mountain quick step Ken Olsen Aggarwal quantitative aptitude book System Guide (Amazing Engine Rule Booklet, Am1/2700) Psychology google ciccarelli books Welcome to the World of Orangutans Run on sentence worksheet 5th grade Export s note to