Appendix C
ISO 9126 Metrics

As we mentioned in the introduction, ISO 9126 lists a number of metrics that could be collected from both static (internal) and dynamic (external) testing. Some of the internal metrics are somewhat theoretical and may be problematic for less-mature organizations (for example, how many bugs were actually found in a review compared to how many were estimated to be there). Many of these metrics would require a very high level of organizational maturity to track. For example, several of them are based on the "number of test cases required to obtain adequate test coverage" from the requirements specification documentation. What is adequate coverage? ISO 9126 does not say. That is up to each organization to decide for themselves. It is Jamie's experience that the answer to that question is often vaporous.

Question: What is adequate? Answer: Enough!

Jamie and Rex have followed different career paths and have different experiences when it comes to static testing. While Jamie has worked for some organizations that did some static testing, they were more the exception than the rule. The ones that did static testing did not track many formal metrics from that testing. Rex, on the other hand, has worked with far more organizations that have performed static testing and utilized internal metrics.

In this appendix, we are going to list the metrics to which ISO 9126 refers. We are not claiming that these will always be useful to track; the people at each organization must make decisions as to which metrics they believe will give them value. We are listing them because the authors of the ISO 9126 standard believed that they can add some value to some organizations sometimes.
1. **ISO 9126 Reliability Metrics**

1.1 **Internal reliability metrics**

These can be used to help predict if the software will satisfy reliability needs of the organization during its development of the software system.

**Maturity** metrics indicate a set of attributes for assessing the maturity of the software.

**Fault detection**: A measure of how many defects\(^1\) were detected in the reviewed software product. This metric is collected by counting the number of detected bugs found in review and comparing that number to the amount that were estimated to be found in this phase of static testing. The metric is calculated by the formula

\[
X = \frac{A}{B}
\]

where \(A\) is the absolute number of bugs detected (from the review report) and \(B\) is the estimated number expected.\(^2\) A high value for this metric implies a good level of product quality.

**Fault removal**: A measure of how many defects that were found during review are removed (corrected) during design and implementation phases. The metric can be calculated by the formula

\[
X = \frac{A}{B}
\]

where \(A\) is the number of bugs fixed during design and coding and \(B\) is the number that were found during review. The closer the value is to 1, the better. A fault removal value of one would mean that every detected defect had been removed.

**Test adequacy**: A measure of how many of the required test cases are covered by the test plan. To calculate this metric, count the number of test cases planned

---

1. Remember that ISTQB treats the terms *bug*, *defect*, and *fault* as synonyms. ISO 9126 appears to use the term *fault* to mean bug, defect, and sometimes failure. Wherever possible, we have tried to reword the standard to meet ISTQB definitions.

2. ISO 9126 uses the term *estimated faults* a number of times. The standard notes that these will tend to come from either past history of the system or a reference model.
(in the test plan) and compare that value to the number of test cases required to "obtain adequate test coverage" using the formula

\[ X = \frac{A}{B} \]

where \( A \) is the number of test cases designed in the test plan and confirmed in review, and \( B \) is the number of test cases required. ISO 9126-3 specifies that the number of test cases needed (\( B \)) should come from the requirements but then does not elucidate exactly how the value is determined. Our research into exactly what determines this value found very little agreement other than ISO 9126 should be modified to define it better.

**Fault tolerance** metrics give us a way to assess the capability of the software to maintain a desired performance level in case of operational failures.

**Failure avoidance**: A measure of how many fault patterns\(^3\) were brought under control to avoid critical and serious failures. To calculate this metric, count the number of avoided fault patterns and compare it to the number of fault patterns to be considered, using the formula

\[ X = \frac{A}{B} \]

where \( A \) is the number of fault patterns that were explicitly avoided in the design and code and \( B \) is the number to be considered as defined by the requirements specification document.

**Incorrect operation avoidance**: A measure of how many functions are implemented with specific designs or code added to prevent the system from incorrect operation. As with failure avoidance, we count the number of functions that have been implemented to avoid or prevent critical and serious failures from occurring and compare them to the number of incorrect operation patterns that have been defined in the requirements. Examples of incorrect operation patterns to be avoided, as given by ISO 9126-3, include accepting incorrect data types as parameters, incorrect sequences of data input, and incorrect sequences

---

\(^3\) ISO 9126 uses this term, *fault patterns*, for which we were not able to find a formal definition, although there is an example in the standard: "Out of range data deadlock." Interestingly enough, this term is used frequently in scientific papers, but is never defined. From context, it would appear a fault pattern is a set of similar type defects (or failures) that have some matching criteria.
of operations. Once again, calculation of the metric is done by using the formula

$$X = \frac{A}{B}$$

where A is the number of incorrect operations that are explicitly designed to be prevented and B is the number to be considered as given in the requirements. ISO 9126-3 does not define exactly what it considers critical or serious failures.

An example of avoidance might be having precondition checks before processing an API call where each argument is checked to make sure it contains the correct data type and permissible values. Clearly, the developers of such a system must make trade-offs between safety and speed of execution.

**Recoverability** metrics are used for assessing the software system's capability to reestablish an adequate level of performance and the ability to recover the data directly affected in case of a failure. There are two of these metrics defined.

**Restorability**: A measure of how capable the system is to restore itself to full operation after an abnormal event or request. This metric is calculated by counting the number of restoration requirements that are implemented by the design and/or code and comparing them to the number in the requirements specification document. For example, if the system has the ability to redo or undo an action, it would be counted as a restoration requirement. Also included are database and transaction checkpoints that allow rollback operations. The formula for the measurement again consists of a ratio,

$$X = \frac{A}{B}$$

where A is the number of restoration requirements that are found in the review documents and B is the number called for in the requirements or design documents. As in the other metrics, the higher this value, the better.

**Restoration effectiveness**: A measure of how effective the restoration techniques will be. Remember that all of these are internal metrics that are expected to come from static testing. According to ISO 9126-3, we get this measurement by calculation and/or simulation. The intent is to find the number of restoration requirements that are expected to be met when those requirements specify a specific time target. Of course, if the requirements do not have a time target, this metric is moot. The measurement has the now-familiar formula of

$$X = \frac{A}{B}$$
where \( A \) is the number of implemented restoration requirements meeting the target restore time and \( B \) is the total number of requirements that have a specified target time. For example, suppose that the requirements specification not only requires the ability to roll back a transaction, but it also defines that it must do so within \( N \) milliseconds once it is triggered. If this capability is actually implemented, and we calculate/simulate that it would actually work correctly, the metric would equal 1/1. If not implemented, the metric would be 0/1.

**Reliability compliance** is involved with assessing the capability of the system to comply with such items as standards, conventions, or regulations of the user organization in relation to reliability. This metric is calculated by using the same ratio,

\[
X = \frac{A}{B}
\]

where \( A \) is the number of regulations we are in compliance with as confirmed by our static testing and \( B \) is the number of total regulations, standards, and conventions that apply to the system.

### 1.2 External reliability metrics

ISO 9126 also describes some interesting external metrics for reliability. Remember that external metrics are defined in ISO 9126-2 and are measured during dynamic testing. The categories of these metrics match the internal ones: maturity, fault tolerance, recoverability, and compliance.

**Maturity** metrics measure the freedom of the software system from failures caused by bugs in the system itself.

**Estimated latent fault density**: A measure of how many defects remain in the system that may emerge as future failures. This metric depends on using a reliability growth estimation model as we discussed earlier in this appendix. The formula for this metric is

\[
X = \frac{\text{abs}(A1 - A2)}{B}
\]

where \( A1 \) is the total number of predicted latent defects in the system, \( A2 \) is the total number of actually occurring failures, and \( B \) is the product size. To get the predicted number of latent defects, you count the number of defects that are detected through testing during a specified trial period and then predict the
number of future failures using the reliability growth model. The actual count of failures found comes from incident reports. Interestingly enough, ISO 9126-2 does not define how size is measured; it should not matter as long as the same measurement is used in the reliability growth model. Some organizations prefer KLOC (thousands of lines of source code); others prefer to measure function points.

There is the very real possibility that the number of actual failures may exceed the number estimated. In that case, ISO 9126-2 recommends reestimating, possibly using a different reliability growth model. The standard also makes the point that a larger number of latent defects should not be estimated simply to make the system look better when fewer failures are actually found. We're sure that could never happen!

**Failure density against test cases:** A measure of how many failures were detected during a defined trial period. In order to calculate this metric, use the formula

\[ X = \frac{A1}{A2} \]

where \( A1 \) is the number of detected failures during the period and \( A2 \) is the number of executed test cases. A desirable value for this metric will depend on the stage of testing. The larger the number in earlier stages (unit, component, and integration testing), the better. The smaller the number in later stages of testing and in operation, the better. This is a metric that is really only meaningful when it is monitored throughout the lifecycle and viewed as a trend rather than snapshots in time. Although ISO 9126-2 does not mention it, the granularity of test cases might skew this metric if there are large differences between test cases at different test levels. The standard does note that testing should include appropriate test cases: normal, exceptional, and abnormal tests.

**Failure resolution:** A measure of how many failure conditions are resolved without reoccurrence. This metric can be calculated by the formula

\[ X = \frac{A1}{A2} \]

where \( A1 \) is the total number of failures that are resolved and never reoccur during the trial period and \( A2 \) is the total number of failures that were detected. Clearly, every organization would prefer that this value be equal to 1. In real life,
however, some failures are not resolved correctly the first time; the more often failures reoccur, the closer to zero this metric will get.

**Fault density**: A measure of how many failures were found during the defined trial period in comparison to the size of the system. The formula is

\[ X = \frac{A}{B} \]

where \( A \) is the number of detected failures and \( B \) is the system size (again, ISO 9126-2 does not define how size is measured). This is a metric where the trend is most important. The later the stage of testing, the lower we would like this metric to be. Two caveats must be made when discussing fault density. Duplicate reports on the same defect will skew the results, as will erroneous reports (where there was not really a defect, but instead the failure was caused by the test environment, bad test case, or other external problem). This metric is a very good measure of the effectiveness of the test cases.

**Fault removal**: A measurement of how many defects have been corrected. There are two components to this metric: one covering the actually found defects and one covering the estimated number of latent defects. The formulae

\[ X = \frac{A_1}{A_2} \]
\[ Y = \frac{A_1}{A_3} \]

are used, where \( A_1 \) is the number of corrected defects, \( A_2 \) is the total number of actually detected defects, and \( A_3 \) is the total number of estimated latent defects in the system. In reality, the first formula is measuring how many found defects are not being removed. If the value is 1, every defect found was removed; the smaller the number, the more defects are being left in the system. As with some of the other metrics, this measurement is more meaningful when viewed as a trend rather than isolated in time.

If the estimated value, \( Y \), is greater than 1, the organization may want to investigate if the software is particularly buggy or if the estimate based on the reliability growth model was faulty. If \( Y \) is appreciably less than 1, the organization may want to investigate if the testing was not adequate to detect all of the defects. Remember that duplicate incident reports will skew this metric.

**Mean time between failures** (MTBF): A measure of how frequently the software fails in operation. To calculate this metric, count the number of failures
that occurred during a defined period of usage and compute the average interval between the failures. This can be calculated two ways:

\[ X = \frac{T1}{A} \]
\[ Y = \frac{T2}{A} \]

T1 is the total operation time, and T2 is the sum of all the time intervals when the system was running. The second formula can be used when there were time during the interval when the system was not running. Whichever formula is used, A is the total number of failures that were observed during the time the system was actually operating. Clearly, it is better the greater the value of X or Y. This metric may require more research as to why the failures occurred. For example, there may be a specific function that is failing while other functionality may be working fine. Determination of the distribution of the types of failures may be valuable, especially if there are different ways to use the system.

Test coverage: A measure of how many test cases have been executed during testing. This metric requires us to estimate how much testing it would require to obtain adequate coverage based on the requirements specification. ISO 9126-2 does not define what that level of coverage is; it is our experience that such a value is often—at best—a wild guess in immature organizations. The formula to calculate this metric is

\[ X = \frac{A}{B} \]

where A is the number of test cases actually executed and B is the estimated number of test cases, from the requirements, that are needed for adequate coverage. The closer this value comes to 1, the better.

Test maturity: A measure of how well the system has been tested. This is used, according to ISO 9126-2, to predict the success rate the system will achieve in future testing. As before, the formula consists of

\[ X = \frac{A}{B} \]

where A is the number of test cases passed and B is the estimated number of test cases needed for adequate coverage based on the requirements specification documentation. In this particular case, the standard does make some recommendation as to the type of testing to be used in this metric. The standard recommends stress testing using live historical data, especially from peak periods.
It further recommends using user operation scenarios, peak stress testing, and overloaded data input. The closer this metric to 1—that is, the more test cases pass in comparison to all that should be run—the better.

**Fault tolerance** external measurements are related to the capability of the system to maintain a specified performance level in cases of operational faults.

**Breakdown avoidance**: A measure of how often the system being tested causes the breakdown of the total production environment. This is calculated using the formula

\[ X = 1 - \frac{A}{B} \]

where \( A \) is the number of total breakdowns and \( B \) is the number of failures. The closer this value is to 1 (i.e., the number of breakdowns is closer to zero), the better. The term *breakdown* is defined as “execution of user tasks is suspended until either the system is restarted or that control is lost until the system is forced to be shut down.” Ideally, a system will be engineered to be able to handle internal failures without causing the total breakdown of the system; this of course usually comes about by adding extra fault sensing and repair code to the system. ISO 9126-2 notes that when few failures do occur, it might make more sense to measure the time between them instead of this metric.

**Failure avoidance**: A measure of how many fault patterns were brought under control to avoid critical and serious failures. As before, the term *fault patterns* is used without a formal definition; however, two examples are given: out of range data or deadlocks. Remember that these metrics are for dynamic testing. To capture this particular metric, we would perform negative tests and then calculate how often the system would be able to survive the forced fault without having it tumble into a critical or serious failure. ISO 9126-2 actually gives us definitions for failure avoidance levels as follows:

- **Critical**: The entire system stops or serious database destruction occurs.
- **Serious**: Important functionality becomes inoperable with no possible workarounds.
- **Average**: Most functions are still available, but limited performance occurs with workarounds.
- **Small**: A few functions experience limited performance with limited operation.
- **None**: Impact does not reach end user.
The metric is calculated using the familiar formula

\[ X = \frac{A}{B} \]

where \( A \) is the number of avoided critical and serious failure occurrences against test cases for a given fault pattern and \( B \) is the number of executed test cases for the fault pattern. A value closer to 1, signifying that the user will suffer fewer critical and serious failures, is better.

**Incorrect operation avoidance**: A measure of how many system functions are implemented with the ability to avoid incorrect operations or damage to data. Incorrect operations that may occur are defined in this case to include the following:

- Incorrect data types as parameters
- Incorrect sequence of data input
- Incorrect sequence of operations

To calculate this measurement, we count the number of negative test cases which fail to cause critical or serious failures (i.e., the negative test cases pass) compared to the number of test cases we executed for that purpose. We use the formula

\[ X = \frac{A}{B} \]

where \( A \) is the number of test cases that pass (i.e., no critical or serious failures occur) and \( B \) is the total number run. The closer to 1, the better this metric shows incorrect operation avoidance.

**Recoverability metrics** should be able to measure the ability of the system to reestablish an adequate level of performance and recover the data directly affected in the case of a failure. When discussing recoverability, ISO 9126-2 specifically is speaking of automated recovery without human intervention. This functionality, obviously, must intentionally be built into the system by adding extra code.

**Availability**: A measure of how available the system is for use during a specified period of time. This is calculated by testing the system in an environment as much like production as possible, performing tests against all system functionality, and uses the formulae
\[ X = \{ \frac{T_o}{T_o + T_r} \} \]
\[ Y = \frac{A1}{A2} \]

where \( T_o \) is the total operation time and \( T_r \) is the time the system takes to repair itself (such that the system is not available for use); \( A1 \) is the total number of test cases that were successful and \( A2 \) is the number of total test cases run.

In the above formulae, \( X \) is the total time available (the closer to 1, the more available the system was) while \( Y \) is a measure of the number of test cases that showed successful availability of the system (again, the closer to 1, the better).

**Mean down time**: A measure of the average time the system remains unavailable when a failure occurs before the system eventually starts itself back up. As we said earlier, ISO 9126-2 is interested in measuring this value only when the restoration of functionality is automatic; this metric is not used when a human has to intervene with the system to restart it. To calculate this measurement, measure the time the system is unavailable after a failure during the specified testing period using the formula

\[ X = \frac{T}{N} \]

where \( T \) is the total amount of time the system is not available and \( N \) is the number of observed times the system goes down. The closer \( X \) is to zero, the better this metric is (i.e., the shorter the unavailable time).

**Mean recovery time**: A measure of the average time it takes for the system to automatically recover after a failure occurs. This metric is measured during a specified test period and is calculated by the formula

\[ X = \frac{\text{Sum}(T)}{N} \]

where \( T \) is the time to recover for each failure and \( N \) is the number of test cases that triggered a failing condition for which recovery occurred. Smaller is clearly better for this measure. ISO 9126-2 notes that this metric may need to be refined to distinguish different types of recovery. For example, the recovery of a destroyed database is liable to take much longer than the recovery of a single transaction. Putting all different kinds of failures into the same measurement may cause it to be misleading.

**Restartability**: A measure of how often a system can recover and restart providing service to users within a target time period. As before, the restartability
metric is concerned only with automatic recovery (where no manual intervention occurs). The target time period likely comes from the requirements (e.g., a specified service-level agreement is in place). To calculate this metric, we count the number of times that the system must restart due to failure during the testing period and how many of those restarts occurred within the target time. We use the formula

\[ X = \frac{A}{B} \]

where \( A \) is the number of restarts that were performed within the target time and \( B \) is the total number of restarts that occurred. The closer to 1, the better. The standard points out that we may want to refine this metric because different types of failures are liable to have radically different lengths of recovery time.

**Restorability**: A measure of how capable the system is in restoring itself to full operation after an abnormal event or request. In this case, we are interested only in the restorations that are defined by the requirements. For example, the requirements may define that the system contain the ability to restore certain operations, including database checkpoints, transaction checkpoints, redo functions, undo functions, etc. This metric is not concerned with other restorations that are not defined in the specifications. Calculation of this metric is done using the formula

\[ X = \frac{A}{B} \]

where \( A \) is the number of restorations successfully made and \( B \) is the number of test cases run based on the requirements. As before, values closer to 1, showing better restorability, are preferable.

**Restore effectiveness**: A measure of how effective the restoration capability of the system is. This metric is calculated using the same old formula

\[ X = \frac{A}{B} \]

where \( A \) is the number of test cases where restoration was successfully completed within the target time and \( B \) is the number of test cases performed. Values closer to 1 show better restoration effectiveness. Once again, target time is likely defined in the requirements or service-level agreement.
And finally, the standard contains a reliability compliance metric that measures how acquiescent the system is to applicable regulations, standards, and conventions. This is measured by using the formula

\[ X = 1 - \frac{A}{B} \]

where A is the number of reliability compliance items that were specified but were not implemented during the testing and B is the total number of reliability compliance items specified. ISO 9126-2 lists the following places where compliance specifications may exist:

- Product description
- User manual
- Specification of compliance
- Related standards, conventions, and regulations

The closer to 1, the better; 1 would represent total compliance with all items.

2. ISO 9126 Efficiency Metrics

2.1 Internal efficiency metrics

There are a number of internal metrics that an organization may wish to collect when static testing for efficiency. As always, the context of the software will determine which of these your project may wish to track.

**Time behavior** metrics define a set of measurements that might be useful for predicting how the computer will respond (timing-wise) during operation. Remember that these are internal metrics, so the metric may be calculated or simulated.

**Response time**: A measure of the estimated time to perform a given task. This measurement estimates the time it will take for a specific action to complete based on the efficiency of the operating system and the application of the system calls. Any of the following might be estimated:

- All (or parts) of design specifications
- Complete transaction path
- Complete modules or parts of the software product
- Complete system during test phase
Clearly, a shorter time is better. The inputs to this measurement are the known characteristics of the operating system added to the estimated time in system calls.

**Throughput time**: A measure of the estimated number of tasks that can be performed over a unit of time. This is measured by evaluating the efficiency of the resources of the system that will be handling the calls as well as the known characteristics of the operating system. The greater this number, the better.

**Turnaround time**: A measure of the estimated time to complete a group of related tasks performing a specific job. As in the other time-related metrics, we need to estimate the operating system calls that will be made and the application system calls involved. The shorter the time, the better. The same entities listed for response time can all be estimated for this metric.

**Resource utilization** metrics indicate a set of attributes for predicting the utilization of hardware.

**I/O utilization**: A measure of the estimated I/O utilization to complete a specified task. The value of this metric is the number of buffers that are expected to be required (calculated or simulated). The smaller this value is, the better. Note that each task will have its own value for this metric.

**I/O utilization message density**: A measure of the number of messages relating to I/O utilization in the lines of code responsible for making system calls. To calculate this value, count the number of error messages pertaining to I/O failure and warnings and compare that to the estimated number of lines of code involved in the system calls using the formula

\[ X = \frac{A}{B} \]

where A is the number of I/O-related error messages and B is the number of lines of code directly related to system calls. The greater this size, the better.

**Memory utilization**: A measure of the estimated memory size that the software system will occupy to complete a specified task. This is a straightforward estimation of the number of bytes; each different task should be estimated. As expected, the smaller this estimated footprint, the better.
Memory utilization message density: A measure of the number of error messages relating to memory usage in the lines of code responsible for making the system calls that are to be used. To calculate this metric, count the number of error messages pertaining to memory failure and warnings and compare that to the number of lines of code responsible for the system calls using the formula

\[ X = \frac{A}{B} \]

where \( A \) is the number of memory-related error and warning messages and \( B \) is the number of lines of code directly related to the system calls. The greater this ratio, the better.

Transmission utilization: A measure of the amount of transmission resources that will likely be needed based on an estimate of the transmission volume for performing tasks. This metric is calculated by estimating the number of bits that will be transmitted by system calls and dividing that by the time needed to perform those calls. As always, because this is an internal value, these numbers must all be either calculated or simulated.

And, finally, ISO 9126-3 defines efficiency compliance as a measure of how compliant the system is estimated to be against applicable regulations, standards, and conventions. To calculate this, we use the formula

\[ X = \frac{A}{B} \]

where \( A \) is the number of items related to efficiency compliance that are judged—in reviews—as being correctly implemented and \( B \) is the total number of compliance items. The closer this value is to one, the more compliant it is.

2.2 External efficiency metrics

ISO 9126-2 also describes several external metrics for efficiency. Remember that these are to be measured during actual dynamic testing or operations. The standard emphasizes that these should be measured over many test cases or intervals and averaged since the measurements fluctuate depending on conditions of use, processing load, frequency of use, number of concurrent users, etc.

Like the internal version, ISO 9126-2 defines several specific terms. These definitions are slightly different than ISO 9126-3, so we will give them here:

- **Response time**: Time needed to get a response after an event is triggered. Response time is equal to processing time plus transmission time. It notes
that response time is only applicable for an interactive system, not for a stand-alone system.

- Processing time: The elapsed time between receiving a message and sending the result. Sometimes, this will include the operating overhead time. In other cases, this might only include the time used by a specified application. Unfortunately, it is left up to the organization using the standard to decide which is which.

- Turnaround time: The amount of time it takes to get a result from a request. This value includes much more than the time required by the system because it often includes many separate activities interacting with the system. For example, for an ATM, the turnaround time interval might start when a card is put in the ATM and end when currency is removed. Between those two end points, many interactions can occur: typing in the PIN, selecting the transaction desired, selecting an account, selecting an amount, etc.

**Time behavior** metrics measure attributes of the system during testing or actual operation.

**Response time**: A measure of the time it takes to complete a specified task. Alternately, how long does it take before the system responds to a specified operation. To measure this, record the time \( T_1 \) a task is requested. Record the time that the task is complete \( T_2 \). Subtract \( T_1 \) from \( T_2 \) to get the measurement.

**Mean time to response**: A measure of the average response time for a task to complete. Note that this metric is meaningful only when it is measured while the task is performed within a specified system load in terms of concurrent tasks and system utilization. To calculate this value, execute a number of scenarios consisting of multiple concurrent tasks to load the system to a specified value. Measure the time it takes to complete the specified tasks. Then calculate the metric using the formula

\[
X = \frac{T_{mean}}{TX_{mean}}
\]

where \( T_{mean} \) is the average time to complete the task (for \( N \) runs) and \( TX_{mean} \) is the required mean time to response. The required mean time to response can be derived from the system specification, from the user expectation of business
needs, or through usability testing to observe the reaction of users. The nearer to 1 this value is, the better.

**Worst case response time**: A measure of the absolute limit on the time required to complete a task. This metric is subjective in that it asks if the user will always get a reply from the system in a time short enough to be tolerable for that user. To perform this measurement, emulate a condition where the system reaches maximum load. Run the application and trigger the task a number of times, measuring the response each time. Calculate the metric using the formula

\[ X = \frac{T_{\text{max}}}{R_{\text{max}}} \]

where \( T_{\text{max}} \) is the maximum time any one iteration of the task took and \( R_{\text{max}} \) is the maximum required response time. The smaller this value, the better.

**Throughput**: A measure of how many tasks can be successfully performed over a given period of time. Notice that these are likely to be different tasks, all being executed concurrently at a given load. To calculate this metric, use the formula

\[ X = \frac{A}{T} \]

where \( A \) is the number of completed tasks and \( T \) is the observational time period. The larger the value, the better. As before, this measurement is most interesting when comparing the mean and worst case throughput values as we did with response time.

**Mean amount of throughput**: The average number of concurrent tasks the system can handle over a set unit of time, calculated using the same formula \((X = \frac{X_{\text{mean}}}{R_{\text{mean}}})\), where \( X_{\text{mean}} \) is the average throughput and \( R_{\text{mean}} \) is the required mean throughput.

**Worst case throughput ratio**: The absolute limit on the system in terms of the number of concurrent tasks it must perform. To calculate it, use the same formula we saw earlier, \((X = \frac{T_{\text{max}}}{R_{\text{max}}})\), where \( T_{\text{max}} \) is the worst-case time of a single task and \( R_{\text{max}} \) is the required throughput.

**Turnaround time**: A measure of the wait time the user experiences after issuing a request to start a group of related tasks until their completion. As we might expect, this is most meaningful when we look at the mean and worst case turnaround times as follows.
Mean time for turnaround: The average wait time the user experiences compared to the required turnaround time as calculated by \( X = \frac{T_{\text{mean}}}{T_{X_{\text{mean}}}} \). This should be calculated at a variety of load levels.

Worst case turnaround time ratio: The absolute acceptable limit on the turnaround time, calculated in the same way we saw before, \( X = \frac{T_{\text{max}}}{R_{\text{max}}} \).

Waiting time: A measure of the proportion of time users spend waiting for the system to respond. We execute a number of different tasks at different load levels and measure the time it takes to complete the tasks. Then, calculate the metric using the formula
\[
X = \frac{T_a}{T_b}
\]
where \( T_a \) is the total time the user spent waiting and \( T_b \) is the actual task time when the system was busy. An efficient system will have a waiting time of close to zero.

Resource utilization metrics allow us to measure the resources that the system consumes during testing or operation. These metrics are usually measured against the required or expected values.

I/O devices utilization: A measure of how much the system uses I/O devices compared to how much it was designed to use. This is calculated using the formula
\[
X = \frac{A}{B}
\]
where \( A \) is the amount of time the devices are occupied and \( B \) is the specified time the system was expected to use them. Less than and nearer to 1 is better.

I/O loading limits: A measure of the absolute limit on I/O utilization that can be used when performing a given function. This is measured when the system is at its maximum rated load. The formula used is
\[
X = \frac{A_{\text{max}}}{R_{\text{max}}}
\]
where \( A_{\text{max}} \) is the maximum number of I/O messages from a given number of runs and \( R_{\text{max}} \) is the required maximum number of messages the system was designed to use. This assumes that the specification of the system has actually defined this number.
**I/O-related errors**: A measure of how often the user encounters I/O type problems. This is measured at the maximum rated load using the formula

\[ X = \frac{A}{T} \]

where \( A \) is the number of warning messages or errors encountered and \( T \) is the user operating time. The smaller this measure, the better.

**Mean I/O fulfillment ratio**: A measure of the number of I/O-related error messages and/or failures over a specified length of time at the maximum load. This is compared to the required mean using the formula

\[ X = \frac{A_{\text{mean}}}{R_{\text{mean}}} \]

where \( A_{\text{mean}} \) is the average number of I/O error messages and failures over a number of runs and \( R_{\text{mean}} \) is the required average number of I/O-related error messages. The lower this value, the better.

**User waiting time of I/O devices utilization**: A measure of the impact of I/O utilization on the waiting time for users. This is a simple measurement of the waiting times required while I/O devices operate. As you might expect, the shorter this waiting time, the better. This should be measured at the rated load.

**Maximum memory utilization**: A measure of the absolute limit on memory required to fulfill a specific function. Despite its name, this measurement actually looks at error messages rather than the number of bytes needed in memory. This is measured at maximum expected load using the formula

\[ X = \frac{A_{\text{max}}}{R_{\text{max}}} \]

where \( A_{\text{max}} \) is maximum number of memory-related error messages (taken from one run of many) and \( R_{\text{max}} \) is the maximum (allowed) number of memory-related error messages. The smaller this value, the better.

**Mean occurrence of memory error**: A measure of the average number of memory-related error messages and failures over a specified length of time and spec-

---

4. This appears to be an awkward translation. It would not make much sense to require a system to average \( N \) number of error messages over a certain period. Perhaps a better term would be *maximum allowed* number of messages. We will use that phrase in future metrics.
ified load on the system. We calculate this metric using the same formula as before,

\[ X = \frac{A_{\text{mean}}}{R_{\text{mean}}} \]

where \( A_{\text{mean}} \) is the average number of memory error messages over a number of runs and \( R_{\text{mean}} \) is the maximum allowed mean number of memory-related error messages. The lower the better.

**Ratio of memory error/time**: A measure of how many memory errors occurred over a given period of time and specified resource utilization. This metric is calculated by running the system at the maximum rated load for a specified amount of time and using the formula

\[ X = \frac{A}{T} \]

where \( A \) is the number of memory-related warning messages and system errors that occurred and \( T \) is the amount of time. The smaller this value, the better.

**Maximum transmission utilization**: A measure of the actual number of transmission-related error messages to the allowed (required) number while running at maximum load. This metric is calculated using the formula

\[ X = \frac{A_{\text{max}}}{R_{\text{max}}} \]

where \( A_{\text{max}} \) is the maximum number of transmission-related error messages (taken from one run of many) and \( R_{\text{max}} \) is the maximum (allowed) number of transmission-related error messages. The smaller this value, the better.

**Media device utilization balancing**: A measure of the degree of synchronization between different media over a set period of time. This metric is derived while the system is running at maximum transmission load. While the application is running, record the number of errors due to transmission failure and warnings. Calculate the metric using the formula

\[ X = \frac{\text{SyncTime}}{T} \]

where SyncTime is the time devoted to a continuous resource and \( T \) is the required time period during which dissimilar media are expected to complete their tasks with synchronization. The smaller this value, the better.
**Mean occurrence of transmission error**: A measure of the average number of transmission-related error messages and failures over a specified length of time and utilization. This is measured while the system is at maximum transmission load. Run the application under test and record the number of errors due to transmission failure and warnings. The calculation uses the formula

\[ X = \frac{A_{\text{mean}}}{R_{\text{mean}}} \]

where \( A_{\text{mean}} \) is the average number of transmission-related error messages and failures over multiple runs and \( R_{\text{mean}} \) is the maximum allowed number as defined earlier. The smaller the number, the better.

**Mean of transmission error per time**: A measure of how many transmission-related error messages were experienced over a set period of time and at a specified resource utilization. This value is measured while the system is running at maximum transmission loading. The application being tested is run and the number of errors due to transmission failures and warning are recorded. The metric is calculated using the formula

\[ X = \frac{A}{T} \]

where \( A \) is the number of warning messages or system failures and \( T \) is the operating time being measured. Smaller is better for this metric.

**Transmission capacity utilization**: A measure of how well the system is capable of performing tasks within the expected transmission capacity. This is measured while executing concurrent tasks with multiple users, observing the transmission capacity, and comparing it to specified values using the formula

\[ X = \frac{A}{B} \]

where \( A \) is the transmission capacity and \( B \) is the specified transmission capacity designed for the software to use. Less than and nearer to is better.

Finally there is an **efficiency compliance** metric. This is a measure of how compliant the efficiency of the product is with respect to applicable regulations, standards, and conventions. Calculation of this metric is done using the formula

\[ X = 1 - \frac{A}{B} \]
where \( A \) is the number of efficiency compliance items that have not been implemented during testing and \( B \) is the total number of efficiency compliance items that have been specified for the product. The closer to 1, the better.

### 3. ISO 9126 Maintainability Metrics

#### 3.1 Internal maintainability metrics

ISO 9126-3 defines a number of internal maintainability metrics that an organization may wish to track. These should help predict the level of effort required when modifying the system.

**Analyzeability** metrics help predict the maintainer’s or user’s spent effort or resources in trying to diagnose deficiencies and causes of failure or for identifying parts to be modified in the system.

**Activity recording**: A measure of how thoroughly the system status is recorded. This is calculated by counting the number of items that are found to be written to the activity log as specified compared to how many items are supposed to be written based on the requirements. The calculation of the metric is made using the formula

\[
X = \frac{A}{B}
\]

where \( A \) is the number of items implemented that actually write to the activity log as specified (as confirmed in review) and \( B \) is the number of items that should be logged as defined in the specifications. The closer this value is to 1, the more complete the logging is.

**Readiness of diagnostic function**: A measure of how thorough the provision for diagnostic functions is. A diagnostic function analyzes a failure and provides an output to the user or log with an explanation of the failure. This metric is a ratio of the implemented diagnostic functions compared to the number of required diagnostic functions in the specifications and uses the same formula to calculate it,

\[
X = \frac{A}{B}
\]
where A is the number of diagnostic functions that have been implemented (as found in review) and B is the required number from the specifications. This metric can also be used to measure failure analysis capability in the system and ability to perform causal analysis.

**Changeability** metrics help in predicting the maintainer’s or user’s effort when trying to implement a specified modification to the system.

**Change recordability**: A measure of how completely changes to specifications and program modules are documented. ISO 9126-3 defines these as change comments in the code or other documentation. This metric is calculated with the formula

$$X = A / B$$

where A is the number of changes that have comments (as confirmed in reviews) and B is the total number of changes made. The value should be $0 \leq X \leq 1$; the closer to 1 the better. A value near 0 indicates poor change control.

**Stability** metrics to help us predict how stable the system will be after modification.

**Change impact**: A measure of the frequency of adverse reactions after modification of the system. This metric is calculated by counting the number of adverse impacts that occur after the system is modified to the actual number of modifications made using the formula

$$X = 1 - A / B$$

where A is the number of adverse impacts and B is the number of modifications made. The closer to 1, the better. Note that, since there could conceivably be multiple adverse conditions coming from a sloppy change-management style, this metric could actually be negative. For example, suppose one change was made, but 3 adverse reactions were noted. The calculation, using this formula would be as follows:

$$X = 1 - 3/1 = -2$$

**Modification impact localization**: A measure of how large the impact of a modification is on the system. This value is calculated by counting the number
of affected variables from the modification and comparing it to the total number of variables in the product using the formula

$$X = \frac{A}{B}$$

where A is the number of affected variables as confirmed in review and B is the total number of variables. The definition of an affected variable is any variable in a line of code or computer instruction that was changed. The closer this value is to zero, the less the impact of modification is likely to be.

**Testability** metrics to help us predict the amount of help the system has built into it for testing purposes.

**Completeness of built-in test function**: A measure of how complete any built-in test capability is. To calculate this, count the number of implemented built-in test capabilities and compare it to how many the specification call for. The formula to use is

$$X = \frac{A}{B}$$

where A is the number of built-in test functions implemented (as confirmed in review) and B is the number required in the specifications. The closer to 1, the more complete.

**Autonomy of testability**: A measure of how independently the software system can be tested. This metric is calculated by counting the number of dependencies on other systems that have been simulated with stubs or drivers and comparing it to the total number of dependencies on other systems. As in many other metrics, the formula

$$X = \frac{A}{B}$$

is used, where A is the number of dependencies that have been simulated using stubs or drivers and B is the total number of dependencies on other systems. The closer to 1, the better. A value of 1 means that all other dependent systems can be simulated so the software can (essentially) be tested by itself.

**Test progress observability**: A measure of how completely the built-in test results can be displayed during testing. This can be calculated using the formula

$$X = \frac{A}{B}$$
where A is the number of implemented checkpoints as confirmed in review and B is the number required in the specifications. The closer to 1, the better.

Finally, **maintainability compliance** is a measure of how compliant the system is estimated to be with regard to applicable regulations, standards, and conventions. The ratio of compliance items implemented (as based on reviews) to those requiring compliance in the specifications is calculated using the formula

\[ X = \frac{A}{B} \]

where A is the correctly implemented items related to maintainability compliance and B is the required number. The closer to 1, the more compliant the system is.

### 3.2 External maintainability metrics

ISO 9126-2 defines external maintainability metrics that an organization may wish to track. These help measure such attributes as the behavior of the maintainer, the user, or the system when the software is maintained or modified during testing or maintenance.

**Analyzability** metrics to measure the effort needed while trying to diagnose the cause of failures or identifying items to be modified.

**Audit trail capability**: A measure of how easy it is for a user (or maintainer) to identify the specific operation that caused a failure. ISO 9126-2 is somewhat abstract in its explanation of how to record this metric: It says to observe the user or maintainer who is trying to resolve failures. This would appear to be at odds with the way of calculating the metric, which is to use the formula

\[ X = \frac{A}{B} \]

where A is the number of data items that are actually logged during the operation and B is the number of data items that should be recorded to sufficiently monitor status of the software during operation. As in many of the other metrics in this standard, an organization must define for itself exactly what the value for B should be. A value for X closer to 1 means that more of the required data is being recorded.

**Diagnostic function support**: A measure of how capable the diagnostic functions are in supporting causal analysis. In other words, can a user or maintainer
identify the specific function that caused a failure? As with the previous metric, the method of application merely says to observe the behavior of the user or maintainer who is trying to resolve failures using diagnostic functions. To calculate, use the formula

\[ X = \frac{A}{B} \]

where \( A \) is the number of failures that can be successfully analyzed using the diagnostic function and \( B \) is the total number of registered failures. Closer to 1 is better.

**Failure analysis capability**: A measure of the ability of users or maintainers to identify the specific operation that caused a failure. To calculate this metric, use the formula

\[ X = 1 - \frac{A}{B} \]

where \( A \) is the number of failures that are still not found and \( B \) is the total number of registered failures. Note that this is really the flip side of the previous metric, **diagnostic function support**. This metric is a measure of how many failures we could not diagnose, while the previous metric is a measure of how many we did.

**Failure analysis efficiency**: A measure of how efficiently a user or maintainer can analyze the cause of failure. This is essentially a measure of the average amount of time used to resolve system failures and is calculated by the formula

\[ X = \frac{\text{Sum}(T)}{N} \]

where \( T \) is the amount of time for each failure resolution and \( N \) is the number of problems resolved. Two interesting notes are included in the standard for this metric. ISO 9126-2 says that only the failures that are successfully resolved should be included in this measurement; however, it goes on to say that failures not resolved should also be measured and presented together. It also points out that person-hours rather than simply hours might be used for calculating this metric so that effort may be measured rather than simply elapsed time.
**Status monitoring capability**: A measure of how easy it is to get monitored data for operations that cause failures during the actual operation of the system. This metric is generated by the formula

\[ X = 1 - \frac{A}{B} \]

where \( A \) is the number of cases for which the user or maintainer failed to get monitor data and \( B \) is the number of cases for which they attempted to get monitored data during operation. The closer to 1, the better.

**Changeability** metrics help measure the effort needed when trying to implement changes to the system.

**Change cycle efficiency**: A measure of how likely a user’s problem can be solved within an acceptable time period. This is measured by monitoring the interaction between the user and the maintainer and recording the time between the initial user’s request and the resolution of their problem. The metric is calculated by using the formula

\[ T_{av} = \frac{\text{Sum}(T_u)}{N} \]

where \( T_{av} \) is the average amount of time, \( T_u \) is the elapsed time for the user between sending the problem report and receiving a revised version, and \( N \) is the number of revised versions sent. The shorter \( T_{av} \) is, the better; however, large numbers of revisions would likely be counterproductive to the organization, so a balance must be struck.

**Change implementation elapse time**: A measure of how easily a maintainer can change the software to resolve a failure. This is calculated by using the formula

\[ T_{av} = \frac{\text{Sum}(T_m)}{N} \]

where \( T_{av} \) is the average time, \( T_m \) is the elapsed time between when a failure is detected and when the failure cause is found, and \( N \) is the number of registered and removed failures. As before, there are two notes. Failures not yet found should be excluded, and effort (in person-hours) may be used instead of elapsed time. Shorter is better.
Modification complexity: This is also a measure of how easily a maintainer can change the software to solve a problem. This calculation is made using the formula

\[ T = \frac{\text{Sum}(A / B)}{N} \]

where \( T \) is the average time to fix a failure, \( A \) is work time spent to change a specific failure, \( B \) is the size of the change, and \( N \) is the number of changes. The size of the change may be the number of code lines changed, the number of changed requirements, the number of changed pages of documentation, etc. The shorter this time, the better. Clearly, an organization should be concerned if the number of changes are excessive.

Parameterized modifiability: A measure of how easily the user or maintainer can resolve a software problem merely by changing a parameter. While ISO 9126 does not explain what it means by changing a parameter, we believe that any change made outside the system would comply with this. For example, we might make more memory available, use a faster system, or reduce the concurrent number of users to resolve a problem. The calculation of this metric uses the formula

\[ X = 1 - \frac{A}{B} \]

where \( A \) is the number of cases for which the maintainer fails to resolve the failure and \( B \) is the number of cases for which the maintainer tried to resolve by changing the parameter. Closer to 1 is better as it shows more success by parameter change.

Software change control capability: A measure of how easily a user can identify a revised version of the software. This is also listed as how easily a maintainer can change the system to solve a problem. It is measure using the formula

\[ X = \frac{A}{B} \]

where \( A \) is the number of items actually written to the change log and \( B \) is the number of change log items planned such that we can trace the software changes. Closer to 1 is better, but if there are few changes made, the value will tend toward zero.

Stability metrics are used to measure unexpected behavior of the system after it is modified.
Change success ratio: A measure of how well the user can operate the software system after maintenance without further failures. There are two formulae that can be used to measure this metric:

\[
X = \frac{N_a}{T_a} \\
y = \left\{\frac{N_a}{T_a}\right\} / \left\{\frac{N_b}{T_b}\right\}
\]

where \(N_a\) is number of cases in which the user encounters failures after the software is changed, \(N_b\) is the number of times the user encounters failures before the software is changed, \(T_a\) is the operation time (a specified observation time) after the software is changed, and \(T_b\) is the time (a specified observation time) before the software is changed. Smaller and closer to zero is better. Essentially, \(X\) and \(Y\) represent the frequency of encountering failures after the software is changed. The specified observation time is used to try to normalize the metric so we can compare release-to-release metrics better. Also, ISO 9126-2 suggests that the organization may want to differentiate between failures that come after repair to the same module/function and failures that occur in other modules/functions.

Modification impact localization: This is also a measure of how well the user can operate the system without further failures after maintenance. The calculation uses the formula

\[
X = \frac{A}{N}
\]

where \(A\) is the number of failures emerging after modification of the system (during a specified period) and \(N\) is the number of resolved failures. The standard suggests that the "chaining" of the failures be tracked. That is, the organization should differentiate between a failure that is attributed to the change for a previous failure and failures that do not appear to be related to the change. Smaller and closer to zero is better.

Testability metrics measure the effort required to test a modified system.

Availability of built-in test function: A measure of how easily a user or maintainer can perform operational testing on a system without additional test facility preparation. This metric is calculated by the formula

\[
X = \frac{A}{B}
\]
where A is the number of cases in which the maintainer can use built-in test functionality and B is the number of test opportunities. The closer to 1, the better.

**Re-test efficiency**: A measure of how easily a user or maintainer can perform operational testing and determine whether the software is ready for release or not. This metric is an average calculated by the formula

\[ X = \frac{\text{Sum}(T)}{N} \]

where T is the time spent to make sure the system is ready for release after a failure is resolved and N is the number of resolved failures. Essentially, this is the average retesting time after failure resolution. Note that nonresolved failures are excluded from this measurement. Smaller is better.

**Test restartability**: A measure of how easily a user or maintainer can perform operational testing with checkpoints after maintenance. This is calculated by the formula

\[ X = \frac{A}{B} \]

where A is the number of test cases in which the maintainer can pause and restart the executing test case at a desired point and B is the number of cases in which executing test cases are paused. The closer to 1, the better.

Finally, **maintainability compliance** metrics measure how close the system adheres to various standards, conventions, and regulations. Compliance is measured using the formula

\[ X = 1 - \frac{A}{B} \]

where A is the number of maintainability compliance items that were not implemented during testing and B is the total number of maintainability compliance items defined. Closer to 1 is better.
4. ISO 9126 Portability Metrics

4.1 Internal portability metrics
Internal portability metrics are used for predicting the effect the software may have on the behavior of the implementer or the system during a porting activity.

Adaptability metrics help predict the impact on the effort to adapt the system to a different environment.

Adaptability of data structures: A measure of how adaptable the product is to data structure changes. This metric is calculated using the formula

$$X = \frac{A}{B}$$

where A is the number of data structures that are correctly operable after adaptation as confirmed in review and B is the total number of data structures needing adaptation. The closer to 1, the better.

Hardware environmental adaptability: A measure of how adaptable the software is to hardware-related environmental change. This metric is specifically concerned with hardware devices and network facilities. The formula

$$X = \frac{A}{B}$$

is used, where A is the number of implemented functions that are capable of achieving required results in specified multiple hardware environments as required (confirmed in review) and B is the total number of functions that are required to have hardware adaptation capability. The closer to 1, the better.

Organizational environment adaptability: A measure of how adaptable the software is to organizational infrastructure change. This metric is calculated by the formula

$$X = \frac{A}{B}$$

where A is the number of implemented functions that are capable of achieving required results in multiple specified organizational and business environments as confirmed in review and B is the total number of functions requiring such adaptability. The closer to 1, the better.
**System software environmental adaptability:** A measure of how adaptable the software product is to system-software-related environmental changes. The standard specifically lists adaptability to operating system, network software, and co-operated application software as being measured. This metric uses the formula

\[ X = \frac{A}{B} \]

where \( A \) is the number of implemented functions that are capable of achieving required results in specified multiple system software environments as confirmed in review and \( B \) is the total number of functions requiring such capability. Closer to 1 is better.

**Porting user friendliness:** A measure of how effortless the porting operations on the project are estimated to be. This metric uses the same formula,

\[ X = \frac{A}{B} \]

where \( A \) is the number of functions being ported that are judged to be easy, as based in review, and \( B \) is the total number of functions that are required to be easy to adapt.

**Installability** metrics help predict the impact on the effort of a user trying to install the software into a specified environment.

**Ease of setup retry:** A measure of how easy it is expected to be to repeat the setup operation. This is calculated using the formula

\[ X = \frac{A}{B} \]

where \( A \) is the number of implemented retry operations for setup (confirmed in review) and \( B \) is the total number of setup operations required. The closer to 1, the better.

**Installation effort:** A measure of the level of effort that will be required for installation of the system. This metric is calculated using the formula

\[ X = \frac{A}{B} \]

where \( A \) is the number of automated installation steps as confirmed in review and \( B \) is the total number of installation steps required. Closer to 1 (i.e., fully automated) is better.
**Installation flexibility**: A measure of how customizable the installation capability is estimated to be. This is calculated using the formula

\[ X = \frac{A}{B} \]

where \( A \) is the number of implemented customizable installation operations as confirmed in review and \( B \) is the total number required. The closer to 1, the more flexible the installation.

**Coexistence** metrics help predict the impact the software may have on other software products sharing the same operational hardware resources.

**Available coexistence**: A measure of how flexible the system is expected to be in sharing its environment with other products without adverse impact on them. The same formula,

\[ X = \frac{A}{B} \]

is used, where \( A \) is the number of entities with which the product is expected to coexist and \( B \) is the total number of entities in the production environment that require such coexistence. Closer to 1 is better.

**Replaceability** metrics help predict the impact the software may have on the effort of a user who is trying to use the software in place of other specified software in a specific environment and context of use.

**Continued use of data**: A measure of the amount of original data that is expected to remain unchanged after replacement of the software. This is calculated using the formula

\[ X = \frac{A}{B} \]

where \( A \) is the number of data items that are expected to be usable after replacement as confirmed in review and \( B \) is the number of old data items that are required to be usable after replacement. The closer to 1, the better.

**Function inclusiveness**: A measure of the number of functions expected to remain unchanged after replacement. This measurement is calculated using the formula

\[ X = \frac{A}{B} \]
where \( A \) is the number of functions in the new software that produce results similar to the results displayed by the same functions in the old software (as confirmed in review) and \( B \) is the number of old functions. The closer to 1, the better.

**Portability compliance** metrics help assess the capability of the software to comply with standards, conventions, and regulations that may apply. It is measured using the formula

\[
X = \frac{A}{B}
\]

where \( A \) is the number of correctly implemented items related to portability compliance as confirmed in review and \( B \) is the total number of compliance items.

### 4.2 External portability metrics

External portability metrics measure such attributes as the behavior of the operator or the system during porting activities.

**Adaptability** metrics measure the behavior of the system or user who is trying to adapt the software to different environments.

**Adaptability of data structures**: A measure of how easily a user or maintainer can adapt software to data in a new environment. This metric is calculated using the formula

\[
X = \frac{A}{B}
\]

where \( A \) is the number of data that are not usable in the new environment because of adaptation limitations and \( B \) is the number of data that were expected to be operable in the new environment. The larger the number (i.e., the closer to 1), the better. The data here include such entities as data files, data tuples,\(^5\) data structures, databases, etc. When this metric is calculated, the same type of data should be used for both \( A \) and \( B \).

---

\(^5\) The best definition we can find for this term is a multidimensional data set. ISO 9126-2 does not give a formal definition.
Hardware environmental adaptability: A measure of how easily the user or maintainer can adapt the software to the environment. This metric is calculated using the formula

\[ X = 1 - \frac{A}{B} \]

where A is the number of tasks that were not completed or did not work to adequate levels during operational testing with the new environment hardware and B is the total number of functions that were tested. The larger (closer to 1), the better. ISO 9126-2 specifies that this metric is to be used in reference to adaptability to hardware devices and network facilities. That separates it from the next metric.

Organizational environment adaptability: A measure of how easily the user or maintainer can adapt software to the environment, specifically the adaptability to the infrastructure of the organization. This metric is calculated much like the previous one, using the formula

\[ X = 1 - \frac{A}{B} \]

where A is the number of tasks that could not be completed or did not meet adequate levels during operational testing in the user’s business environment and B is the total number of functions that were tested. This particular metric is concerned with the environment of the business operation of the user’s organization. This separates it from the next, similar measure. Larger is better.

System software environmental adaptability: This is also a measure of how easily the user or maintainer can adapt software to the environment, specifically adaptability to the operating system, network software, and co-operated application software. This metric is also calculated using the same formula,

\[ X = 1 - \frac{A}{B} \]

where A is the number of tasks that were not completed or did not work to adequate levels during operational testing with operating system software or concurrently running application software and B is the total number of functions that were tested. Again, larger is better.

Porting user friendliness: This final adaptability metric also is a measure of how easily a user can adapt software to the environment. In this case it is cal-
culated by adding up all of the time that is spent by the user to complete adaptation of the software to the user’s environment when the user attempts to install or change the setup.

**Installability** metrics measure the behavior of the system or user who is trying to install the software into a specified environment.

**Ease of installation**: A measure of how easily a user can install software to the operational environment. This is calculated by the formula

\[
X = \frac{A}{B}
\]

where \(A\) is the number of cases in which a user succeeded in changing the install operation for their own convenience and \(B\) is the total number of cases in which a user tried to change the install procedure. The closer to 1, the better.

**Ease of setup retry**: A measure of how easily a user can retry the setup installation of the software. The standard does not address exactly why the retry might be needed; just that the retry is attempted. The metric is calculated using the formula

\[
X = 1 - \frac{A}{B}
\]

where \(A\) is the number of cases where the user fails in retrying the setup and \(B\) is the total number of times attempted. The closer to 1, the better.

**Coexistence** metrics measure the behavior of the system or user who is trying to use the software with other independent software in a common environment sharing common resources.

**Available coexistence**: A measure of how often a user encounters constraints or unexpected failures when operating concurrently with other software. This is calculated using the formula

\[
X = \frac{A}{T}
\]

where \(A\) is the number of constraints or failures that occur when operating concurrently with other software and \(T\) is the time duration of operation. The closer to zero, the better.

**Replaceability** metrics measure the behavior of the system or user who is trying to use the software in place of other specified software.
Continued use of data: A measure of how easily a user can continue to use the same data after replacing the software. Essentially, this metric measures the success of the software migration. The formula

\[ X = \frac{A}{B} \]

is used, where A is the number of data items that are able to be used continually after software replacement and B is the number of data items that were expected to be used continuously. The closer to 1, the better. This metric can be used for a new version of the software or for a completely new software package.

Function inclusiveness: A measure of how easily the user can continue to use similar functions after replacing a software system. This metric is calculated the same way, using the formula

\[ X = \frac{A}{B} \]

where A is the number of functions that produce similar results in the new software where changes have not been required and B is the number of similar functions provided in the new software as compared to the old. The closer this value is to 1, the better.

User support functional consistency: A measure of how consistent the new components are to the existing user interface. This is measured by using the formula

\[ X = 1 - \frac{A}{B} \]

where A is the number of functions found by the user to be unacceptably inconsistent to that user's expectation and B is the number of new functions. Larger is better, meaning that few new functions are seen as inconsistent.

Portability compliance metrics measure the number of functions that fail to comply with required conventions, standards, or regulations. This metric uses the formula

\[ X = 1 - \frac{A}{B} \]

where A is the number of portability compliance items that have not been implemented and B is the total number of portability compliance items that are specified. The closer to one, the better. ISO 9126-2 notes that this is a metric
that works best when seen as a trend, with increasing compliance coming as the system becomes more mature.