

**PS.A Form template for Reporting Software Problems**

Originator:	SPR No.:
	Date:
1. Software Item Title:	
2. Software Item Version/Release No.:	
3. Priority: Critical/Urgent/Routine (underline choice)	
4. Problem Description:	
5. Description of Environment:	
6. Recommended Solution:	
7. Review Decision: Close/Update/Action/Reject (underline choice)	
8. Review Decision: Close/Update/Action/Reject (underline choice)	

**PS.B Form template for Requesting Change to Software**

Originator:	SRC No.:
Related SPRs:	Date:
1. Software Item Title:	
2. Software Item Version/Release Number:	
3. Priority: Critical/Urgent/Routine (underline choice)	
4. Changes Required:	
5. Responsible Staff:	
6. Estimated Start Date, End Date and Manpower Report:	
7. Attachments:	

## PS.C Guidelines on how to Evaluate Software

- Performance;
- resource consumption;
- cohesion;
- coupling;
- complexity;
- consistency;
- portability;
- reliability;
- maintainability;
- safety;
- security

The effect of changes may be evaluated with the aid of reverse engineering tools and librarian tools. Reverse engineering tools can identify the modules affected by a change at program design level (e.g. identify each module that uses a particular global variable). Librarian tools with cross-reference facilities can track dependencies at the source file level (e.g. identify every file that includes a specific file).

There is often more than one way of changing the software to solve a problem, and software engineers should examine the options, compare their effects on the software, and select the best solution. The following sections provide guidance on the evaluation of the software in terms of the attributes listed above.

### PS.C.1 Performance

Performance requirements specify the capacity and speed of the operations the software has to perform. Design specifications may specify how fast a component has to execute.

The effects of a change on software performance should be predicted and later measured in tests. Performance analysis tools may assist measurement. Prototyping may be useful.

### PS.C.2 Resource Consumption

Resource requirements specify the maximum amount of computer resources the software can use. Design specifications may specify the maximum resources available.

The effects of a change on resource consumption should be predicted when it is designed and later measured in tests. Performance analysis tools

and system monitoring tools should be used to measure resource consumption. Again, prototyping may be useful.

### PS.C.3 Cohesion

Cohesion measures the degree to which the activities within a component relate to one another. The cohesion of a software component should not be reduced by a change. Cohesion effects should be evaluated when changes are designed.

The *Guide to the software Architectural Design Phase* identifies seven types of cohesion ranging from functional (good) to coincidental (bad). This scale can be used to evaluate the effect of a change on the cohesion of a software component.

### PS.C.4 Coupling

Coupling measures the interdependence of two or more components. Changes that increase coupling should be avoided, as they reduce information hiding. Coupling effects should be evaluated when changes are designed.

The *Guide to the Software Architectural Design Phase* identifies five types of coupling ranging from 'data coupling' (good) to 'content coupling' (bad). This scale can be used to evaluate the effect of a change on the coupling of a software component.

### PS.C.5 Complexity

During the operations and maintenance phase the complexity of software naturally tends to grow because its control structures have to be extended to meet new requirements. Software engineers should contain this natural growth of complexity because more complex software requires more testing, and more testing requires more effort. Eventually changes become infeasible because they require more effort than is available to implement them. Reduced complexity means greater reliability and maintainability.

Software complexity can be measured by several metrics, the best known metric being cyclomatic complexity. Procedures for measuring cyclomatic complexity and other important metrics are contained in the *Guide to the Software Architectural Design Phase* and the *Guide to Software Verification and Validation*. McCabe has proposed an 'essential complexity metric' for measuring the distortion of the control structure of a module caused by a software change.

### PS.C.6 Consistency

Coding standards define the style in which a programming language should be used. They may enforce or ban the use of language features and define rules for layout and presentation. Changes to software should conform to the coding standards and should be seamless.

Polymorphism means that a function will perform the same operation on a variety of data types. Programmers extending the range of data types can easily create inconsistent variation of the same function. This can cause a polymorphic function to behave in unexpected ways. Programmers modifying a polymorphic function should fully understand what the variations of the function do.

#### PS.C.7 Portability

Portability is measured by the ease of moving software from one environment to another. Portability can be achieved by adhering to language and coding standards. A common way to make software portable is to encapsulate platform-specific code in 'interface modules'. Only the interface modules need to be modified when the software is ported.

Software engineers should evaluate the effect of a modification on the portability of the software when it is designed. Changes that reduce portability should be avoided.

#### PS.C.8 Reliability

Reliability is most commonly measured by the Mean Time Between Failures (MTBF). Changes that reduce reliability should be avoided.

Changes that introduce defects into the software make it less reliable. The software verification process aims to detect and remove defects. Walkthroughs and inspections should be used to verify that defects are not introduced when the change is designed. Tests should be used to verify that no defects have been introduced by the change after it has been implemented.

The effect of a modification on software reliability can be estimated indirectly by measuring its effect on the complexity of the software. This effect can be measured when the change is designed.

Reliability can be reduced by reusing components that have not been developed to the same standards as the rest of the software. Software engineers should find out how reliable a software component is before reusing it.

#### PS.C.9 Maintainability

Maintainability measures the ease with which software can be maintained. The most common maintainability metric is 'Mean Time to Repair'. Changes that make software less maintainable should be avoided.

Examples of changes that make software less maintainable are those that:

- violate coding standards;
- reduce cohesion;
- increase coupling;
- increase essential complexity;

The costs and benefits of changes that make software more maintainable should be evaluated before such changes are made. All modified code must be tested, and the cost of retesting the new code may outweigh the reduced effort required to make future changes.

#### PS.C.10 Safety

Changes to the software should not endanger people or property during operations or following a failure. The effect on the safety of the software should first be evaluated when the change is designed and later during the verification process. The behaviour of the software after a failure should be analyzed from the safety point of view.

#### PS.C.11 Security

Changes to the software should not expose the system to threats to its confidentiality, integrity and availability. The effect of a change on the security of the software should first be evaluated when the change is designed and later during the verification process.

## **PS.D Guidelines on how Modified Software should be Tested**

Modified software must be retested before release. This is normally done by:

- unit, integration and system testing each change;
- running regression tests at unit, integration and system level to verify that there are no side-effects;

Changes that do not alter the control flow should be tested by re-running the white box tests that execute the part of the software that changed. Changes that do alter the control flow should be tested by designing white box tests to execute every new branch in the control flow that has been created. Black box tests should be designed to verify any new functionality.

Ideally system tests should be run after each change. This may be a very costly exercise for a large system, and an alternative less costly approach often adopted for large systems is to accumulate changes and then run the system tests (including regression tests) just before release. The disadvantage of this approach is that it may be difficult to identify which change, if any, is the cause of a problem in the system tests.

**PS.E Form template for the Software Release Note**

Originator:	SRN No.:
Approved by:	Date:
1. Software Item Title:	
2. Software Item Version/Release No.:	
3. Changes in this Release:	
4. Configuration Items included in this Release:	
5. Installation Instructions:	