0.		WG 23 N0660 – 13 Jun 2016		
1.	Category	JSF Rule Number	Current 24772 references	Recommended 24772 references
1.	Code Size and Complexity	Any one function (or method) will contain no more than 200 logical source lines of code (L-SLOCs).		Add 6.31
2.		There shall not be any self-modifying code.	6.49	
3.		All functions shall have a cyclomatic complexity number of 20 or less.		
4.	Breaking Rules	To break a “should” rule, the following approval must be received by the developer: approval from the software engineering lead (obtained by the unit approval in the developmental CM tool)		
5.		To break a “will” or a “shall” rule, the following approvals must be received by the developer: approval from the software engineering lead (obtained by the unit approval in the developmental CM tool) approval from the software product manager (obtained by the unit approval in the developmental CM tool)		
6.		Each deviation from a “shall” rule shall be documented in the file that contains the deviation). Deviations from this rule shall not be allowed, AV Rule 5 notwithstanding.		
7.		Approval will not be required for a deviation from a “shall” or “will” rule that complies with an exception specified by that rule.		
8.	Language	All code shall conform to ISO/IEC 14882:2002(E) standard C++.	6.59	
9.	Character Sets	Only those characters specified in the C++ basic source character set will be used. This set includes 96 characters.		
10.		Values of character types will be restricted to a defined and documented subset of ISO 10646-1.		
11.		Trigraphs will not be used.	6.59	
12.		The following digraphs will not be used: <%, :>, %>, %:, <:		
13.		Multi-byte characters and wide string literals will not be used.		Add 6.58
14.		Literal suffixes shall use uppercase rather than lowercase letters.		
15.	Run-Time Checks	Provision shall be made for run-time checking (defensive programming).	6.8 6.9 6.10 6.15 6.16	Add 6.53
16.	Libraries	Only DO-178B level A [15] certifiable or SEAL 1	6.47	

		C/C++ libraries shall be used with safety-critical (i.e. SEAL 1) code.		
17.	Standard Libraries	The error indicator errno shall not be used.	6.56 6.57 6.58	Add 6.36
18.		The macro offsetof, in library <stddef.h>, shall not be used.	6.47 6.56 6.57 6.58	Remove 6.47
19.		<locale.h> and the setlocale function shall not be used.	6.47 6.56 6.57 6.58	Remove 6.47
20.		The setjmp macro and the longjmp function shall not be used.	6.31 6.32 6.47 6.56 6.57 6.58	Remove 6.32 Remove 6.47
21.		The signal handling facilities of <signal.h> shall not be used.	6.47 6.56 6.57 6.58	Remove 6.47
22.		The input/output library <stdio.h> shall not be used.	6.47 6.56 6.57 6.58	Remove 6.47
23.		The library functions atof, atoi and atol from library <stdlib.h> shall not be used.	6.47 6.56 6.57 6.58	Remove 6.47
24.		The library functions abort, exit, getenv and system from library <stdlib.h> shall not be used.	6.37 6.47 6.56 6.57 6.58	Remove 6.47
25.		The time handling functions of library <time.h> shall not be used.	6.8 6.47 6.56 6.57 6.58	Remove 6.47
26.	Pre-Processing Directives	Only the following pre-processor directives shall be used: #ifndef, #define, #endif, #include	6.52	
27.	#ifndef and #endif Pre-Processing Directives	#ifndef, #define and #endif will be used to prevent multiple inclusions of the same header file. Other techniques to prevent the multiple inclusions of header files will not be used.	6.52	
28.		The #ifndef and #endif pre-processor directives will only be used as defined in AV Rule 27 to prevent multiple inclusions of the same header file.	6.52	
29.	#define Pre-Processing Directive	The #define pre-processor directive shall not be used to create inline macros. Inline functions shall be used instead.	6.52	Add 6.31
30.		The #define pre-processor directive shall not be used to define constant values. Instead, the const qualifier shall be applied to variable declarations to specify constant values.	6.52	
31.		The #define pre-processor directive will only be used as part of the technique to prevent multiple inclusions of the same header file.	6.52	
32.	#include Pre-Processing Directive	The #include pre-processor directive will only be used to include header (*.h) files.	6.52	
33.	Header Files	The #include directive shall use the <filename.h> notation to include header files.		

34.		Header files should contain logically related declarations only.		Add 6.31
35.		A header file will contain a mechanism that prevents multiple inclusions of itself.		
36.		Compilation dependencies should be minimized when possible.		
37.		Header (include) files should include only those header files that are required for them to successfully compile. Files that are only used by the associated .cpp file should be placed in the .cpp file—not the .h file.		
38.		Declarations of classes that are only accessed via pointers (*) or references (&) should be supplied by forward headers that contain only forward declarations.		
39.		Header files (*.h) will not contain non-const variable definitions or function definitions. (See also AV Rule 139.)		
40.	Implementation Files	Every implementation file shall include the header files that uniquely define the inline functions, types, and templates used.		Add 6.41
41.	Style	Source lines will be kept to a length of 120 characters or less.		
42.		Each expression-statement will be on a separate line.		
43.		Tabs should be avoided.		
44.		All indentations will be at least two spaces and be consistent within the same source file.		
45.	Naming Identifiers	All words in an identifier will be separated by the '_' character.		
46.		User-specified identifiers (internal and external) will not rely on significance of more than 64 characters.	7.10	Add 6.17
47.		Identifiers will not begin with the underscore character '_'.		Add 6.17
48.		Identifiers will not differ by: Only a mixture of case The presence/absence of the underscore character The interchange of the letter 'O', with the number '0' or the letter 'D' The interchange of the letter 'I', with the number '1' or the letter 'l' The interchange of the letter 'S' with the number '5' The interchange of the letter 'Z' with the number 2 The interchange of the letter 'n' with the letter 'h'.	6.17	
49.		All acronyms in an identifier will be composed of	6.17	

		uppercase letters.		
50.		The first word of the name of a class, structure, namespace, enumeration, or type created with typedef will begin with an uppercase letter. All others letters will be lowercase.	6.17	
51.		All letters contained in function and variable names will be composed entirely of lowercase letters.	6.17 7.10	
52.		Identifiers for constant and enumerator values shall be lowercase.	6.17	
53.	Naming Files	Header files will always have a file name extension of ".h". 53.1: The following character sequences shall not appear in header file names: ‘, \, /*, //, or ".	7.10	Add 53.1 to 6.57
54.		Implementation files will always have a file name extension of ".cpp".	7.10	
55.		The name of a header file should reflect the logical entity for which it provides declarations.	7.10	
56.		The name of an implementation file should reflect the logical entity for which it provides definitions and have a “.cpp” extension (this name will normally be identical to the header file that provides the corresponding declarations.)	7.10	
57.	Classes	The public, protected, and private sections of a class will be declared in that order (the public section is declared before the protected section which is declared before the private section).		
58.	Functions	When declaring and defining functions with more than two parameters, the leading parenthesis and the first argument will be written on the same line as the function name. Each additional argument will be written on a separate line (with the closing parenthesis directly after the last argument).		
59.	Blocks	The statements forming the body of an if, else if, else, while, do…while or for statement shall always be enclosed in braces, even if the braces form an empty block.	6.28	
60.		Braces ("{}") which enclose a block will be placed in the same column, on separate lines directly before and after the block.		
61.		Braces ("{}") which enclose a block will have nothing else on the line except comments (if necessary).		
62.	Pointers and References	The dereference operator ‘*’ and the address-of operator ‘&’ will be directly connected with the type-specifier.		
63.	Miscellaneous	Spaces will not be used around ‘.’ or ‘->’, nor		

		between unary operators and operands.		
64.	Class Interfaces	A class interface should be complete and minimal.		
65.	Considerations Regarding Access Rights	A structure should be used to model an entity that does not require an invariant.		
66.		A class should be used to model an entity that maintains an invariant.		
67.		Public and protected data should only be used in structs—not classes.		
68.	Member Functions	Unneeded implicitly generated member functions shall be explicitly disallowed.		
69.	const Member Functions	A member function that does not affect the state of an object (its instance variables) will be declared const.		
70.	Friends	A class will have friends only when a function or object requires access to the private elements of the class, but is unable to be a member of the class for logical or efficiency reasons. 70.1: An object shall not be improperly used before its lifetime begins or after its lifetime ends.		Add 70.1 to 6.57
71.	Constructors	Calls to an externally visible operation of an object, other than its constructors, shall not be allowed until the object has been fully initialized. 71.1: A class's virtual functions shall not be invoked from its destructor or any of its constructors.	6.22	
72.		The invariant2 for a class should be: a part of the postcondition of every class constructor, a part of the precondition of the class destructor (if any), • a part of the precondition and postcondition of every other publicly accessible operation.		Add 6.43
73.		Unnecessary default constructors shall not be defined.		
74.		Initialization of nonstatic class members will be performed through the member initialization list rather than through assignment in the body of a constructor.		
75.		Members of the initialization list shall be listed in the order in which they are declared in the class.		
76.		A copy constructor and an assignment operator shall be declared for classes that contain pointers to data items or nontrivial destructors.	6.39	
77.		A copy constructor shall copy all data members and bases that affect the class invariant (a data element representing a cache, for example, would not need to be copied).	6.39	

		77.1: The definition of a member function shall not contain default arguments that produce a signature identical to that of the implicitly-declared copy constructor for the corresponding class/structure.		
78.	Destructors	All base classes with a virtual function shall define a virtual destructor.	6.42	Remove 6.42 Add 6.57
79.		All resources acquired by a class shall be released by the class's destructor.	6.42	Add 6.40, 7.8 Remove 6.42
80.	Assignment Operators	The default copy and assignment operators will be used for classes when those operators offer reasonable semantics.	6.39 6.42	Remove 6.42
81.		The assignment operator shall handle self-assignment correctly	6.42	Remove 6.42
82.		An assignment operator shall return a reference to *this.		
83.		An assignment operator shall assign all data members and bases that affect the class invariant (a data element representing a cache, for example, would not need to be copied).		
84.	Operator Overloading	Operator overloading will be used sparingly and in a conventional manner.	6.55	
85.		When two operators are opposites (such as == and !=), both will be defined and one will be defined in terms of the other.		
86.		Concrete types should be used to represent simple independent concepts.	6.42 6.55	Remove 6.42
87.		Hierarchies should be based on abstract classes.	6.42	
88.		Multiple inheritance shall only be allowed in the following restricted form: n interfaces plus m private implementations, plus at most one protected implementation. 88.1: A stateful virtual base shall be explicitly declared in each derived class that accesses it.	6.42 6.55	
89.		A base class shall not be both virtual and non-virtual in the same hierarchy.	6.42 6.43	
90.		Heavily used interfaces should be minimal, general and abstract.	6.42	Remove 6.42
91.		Public inheritance will be used to implement "is-a" relationships.	6.42 6.43	
92.		A subtype (publicly derived classes) will conform to the following guidelines with respect to all classes involved in the polymorphic assignment of different subclass instances to the same variable or parameter during the execution of the system: Preconditions of derived methods must be at least as weak as the preconditions of the methods they	6.42 6.43	

		override. Postconditions of derived methods must be at least as strong as the postconditions of the methods they override.		
93.		“has-a” or “is-implemented-in-terms-of” relationships will be modeled through membership or non-public inheritance.	6.42 6.43	
94.		An inherited nonvirtual function shall not be redefined in a derived class.	6.42	
95.		An inherited default parameter shall never be redefined.	6.42	
96.		Arrays shall not be treated polymorphically.	6.42	Add 6.45 Remove 6.42
97.		Arrays shall not be used in interfaces. Instead, the Array class should be used. 97.1: Neither operand of an equality operator (== or !=) shall be a pointer to a virtual member function.	6.42 6.55	Remove 6.42
98.	Namespaces	Every nonlocal name, except main(), should be placed in some namespace.		
99.		Namespaces will not be nested more than two levels deep.		
100.		Elements from a namespace should be selected as follows: using declaration or explicit qualification for few (approximately five) names, • using directive for many names.		
101.	Templates	Templates shall be reviewed as follows: with respect to the template in isolation considering assumptions or requirements placed on its arguments. 2. with respect to all functions instantiated by actual arguments.	6.41	
102.		Template tests shall be created to cover all actual template instantiations.	6.41	
103.		Constraint checks should be applied to template arguments.	6.41	
104.		A template specialization shall be declared before its use.	6.41	
105.		A template definition’s dependence on its instantiation contexts should be minimized.	6.41	
106.		Specializations for pointer types should be made where appropriate.		
107.	Function Declaration, Definition and Arguments	Functions shall always be declared at file scope.		Add 6.34

108.		Functions with variable numbers of arguments shall not be used.	6.34	
109.		A function definition should not be placed in a class specification unless the function is intended to be inlined.		
110.		Functions with more than 7 arguments will not be used.		
111.		A function shall not return a pointer or reference to a non-static local object.		Add 6.14
112.		Function return values should not obscure resource ownership.		Add 7.8
113.	Return Types and Values	Functions will have a single exit point.	6.31	Add 7.8
114.		All exit points of value-returning functions shall be through return statements.		Add 6.31, 6.57
115.		If a function returns error information, then that error information will be tested.	6.36	
116.	Function Parameters (Value, Pointer or Reference)	Small, concrete-type arguments (two or three words in size) should be passed by value if changes made to formal parameters should not be reflected in the calling function.	6.32	
117.		Arguments should be passed by reference if NULL values are not possible: 117.1: An object should be passed as const T& if the function should not change the value of the object. 117.2: An object should be passed as T& if the function may change the value of the object.		Add 6.32
118.		Arguments should be passed via pointers if NULL values are possible: 118.1: An object should be passed as const T* if its value should not be modified. 118.2: An object should be passed as T* if its value may be modified.		Add 6.32
119.	Function Invocation	Functions shall not call themselves, either directly or indirectly (i.e. recursion shall not be allowed).	6.35	Add 7.8
120.	Function Overloading	Overloaded operations or methods should form families that use the same semantics, share the same name, have the same purpose, and that are differentiated by formal parameters.	6.20	Remove 6.20
121.	Inline Functions	Only functions with 1 or 2 statements should be considered candidates for inline functions.		
122.		Trivial accessor and mutator functions should be inlined.		
123.		The number of accessor and mutator functions should be minimized.		
124.		Trivial forwarding functions should be inlined.		

125.		Unnecessary temporary objects should be avoided.		
126.	Comments	Only valid C++ style comments (//) shall be used.		
127.		Code that is not used (commented out) shall be deleted.	6.26 7.2	
128.		Comments that document actions or sources (e.g. tables, figures, paragraphs, etc.) outside of the file being documented will not be allowed.		
129.		Comments in header files should describe the externally visible behavior of the functions or classes being documented.		
130.		The purpose of every line of executable code should be explained by a comment, although one comment may describe more than one line of code.		
131.		One should avoid stating in comments what is better stated in code (i.e. do not simply repeat what is in the code).		
132.		Each variable declaration, typedef, enumeration value, and structure member will be commented.		
133.		Every source file will be documented with an introductory comment that provides information on the file name, its contents, and any program-required information (e.g. legal statements, copyright information, etc).		
134.		Assumptions (limitations) made by functions should be documented in the function's preamble.		
135.	Declarations and Definitions	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.	6.20	
136.		Declarations should be at the smallest feasible scope.	6.20	
137.		All declarations at file scope should be static where possible.	6.20	
138.		Identifiers shall not simultaneously have both internal and external linkage in the same translation unit.		
139.		External objects will not be declared in more than one file.		
140.		The register storage class specifier shall not be used.		
141.		A class, structure, or enumeration will not be declared in the definition of its type.		Add 6.5
142.	Initialization	All variables shall be initialized before use. (See also AV Rule 136, AV Rule 71, and AV Rule 73, and AV Rule 143 concerning declaration scope, object construction, default constructors, and the point of variable introduction respectively.)		Add 6.22

143.		Variables will not be introduced until they can be initialized with meaningful values. (See also AV Rule 136, AV Rule 142, and AV Rule 73 concerning declaration scope, initialization before use, and default constructors respectively.)	6.22	
144.		Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.		Add 6.22
145.		In an enumerator list, the '=' construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.		Add 6.5, 6.22
146.	Types	Floating point implementations shall comply with a defined floating point standard. The standard that will be used is the ANSI/IEEE Std 754 [1].	6.4	
147.		The underlying bit representations of floating point numbers shall not be used in any way by the programmer.	6.3 6.4 6.22	Remove 6.22
148.		Enumeration types shall be used instead of integer types (and constants) to select from a limited series of choices.	6.2 6.27	Add 6.5
149.	Constants	Octal constants (other than zero) shall not be used.		
150.		Hexadecimal constants will be represented using all uppercase letters.		
151.		Numeric values in code will not be used; symbolic values will be used instead. 151.1: A string literal shall not be modified. Note that strictly conforming compilers should catch violations, but many do not.	7.3	
152.	Variables	Multiple variable declarations shall not be allowed on the same line.		
153.	Unions and Bit Fields	Unions shall not be used.	6.38	
154.		Bit-fields shall have explicitly unsigned integral or enumeration types only.	6.3	
155.		Bit-fields will not be used to pack data into a word for the sole purpose of saving space. Note: Bit-packing should be reserved for use in interfacing to hardware or conformance to communication protocols.	6.3	
156.		All the members of a structure (or class) shall be named and shall only be accessed via their names.		
157.	Operators	The right hand operand of a && or || operator shall not contain side effects.	6.24	
158.		The operands of a logical && or || shall be parenthesized if the operands contain binary operators.	6.24	

159.		Operators ||, &&, and unary & shall not be overloaded.		Add 6.57
160.		An assignment expression shall be used only as the expression in an expression statement.	6.25	
161.		(doesn't exist)		
162.		Signed and unsigned values shall not be mixed in arithmetic or comparison operations.		Add 6.2
163.		Unsigned arithmetic shall not be used.		
164.		The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the left-hand operand (inclusive). 164.1: The left-hand operand of a right-shift operator shall not have a negative value.	6.9 6.15 6.16	
165.		The unary minus operator shall not be applied to an unsigned expression.		
166.		The sizeof operator will not be used on expressions that contain side effects.		Add 6.58
167.		The implementation of integer division in the chosen compiler shall be determined, documented and taken into account.		
168.		The comma operator shall not be used.		
169.	Pointers & References	Pointers to pointers should be avoided when possible.		Add 6.40
170.		More than 2 levels of pointer indirection shall not be used.		Add 6.40
171.		Relational operators shall not be applied to pointer types except where both operands are of the same type and point to: the same object, the same function, members of the same object, or elements of the same array (including one past the end of the same array).		Add 6.56
172.		(doesn't exist)		
173.		The address of an object with automatic storage shall not be assigned to an object which persists after the object has ceased to exist.	6.33	Add 6.14, 6.18, 6.40
174.		The null pointer shall not be de-referenced.	6.13	Add 6.57
175.		A pointer shall not be compared to NULL or be assigned NULL; use plain 0 instead.		
176.		A typedef will be used to simplify program syntax when declaring function pointers.		
177.	Type Conversions	User-defined conversion functions should be avoided.		Add 6.2, 6.6
178.		Down casting (casting from base to derived class) shall only be allowed through one of the following mechanism:		

		Virtual functions that act like dynamic casts (most likely useful in relatively simple cases) • Use of the visitor (or similar) pattern (most likely useful in complicated cases)		
179.		A pointer to a virtual base class shall not be converted to a pointer to a derived class.		
180.		Implicit conversions that may result in a loss of information shall not be used.		Add 6.2, 6.6
181.		Redundant explicit casts will not be used.		Add 6.2
182.		Type casting from any type to or from pointers shall not be used.	6.11	Add 6.2, 6.38, 6.57
183.		Every possible measure should be taken to avoid type casting.	6.2 6.11 6.38	
184.		Floating point numbers shall not be converted to integers unless such a conversion is a specified algorithmic requirement or is necessary for a hardware interface.	6.4	Add 6.2, 6.6
185.		C++ style casts (const_cast, reinterpret_cast, and static_cast) shall be used instead of the traditional C-style casts.		
186.	Flow Control Structures	There shall be no unreachable code.	6.26	
187.		All non-null statements shall potentially have a side-effect.		Add 6.26
188.		Labels will not be used, except in switch statements.		Add 6.31
189.		The goto statement shall not be used.	6.31	
190.		The continue statement shall not be used.	6.31	
191.		The break statement shall not be used (except to terminate the cases of a switch statement).	6.31	
192.		All if, else if constructs will contain either a final else clause or a comment indicating why a final else clause is not necessary.	6.28	
193.		Every non-empty case clause in a switch statement shall be terminated with a break statement.	6.27	
194.		All switch statements that do not intend to test for every enumeration value shall contain a final default clause.	6.27	
195.		A switch expression will not represent a Boolean value.	6.27	
196.		Every switch statement will have at least two cases and a potential default.	6.27	
197.		Floating point variables shall not be used as loop counters.	6.4	
198.		The initialization expression in a for loop will perform no actions other than to initialize the value of a single for loop parameter. Note that the		Add 6.29

		initialization expression may invoke an accessor that returns an initial element in a sequence:
		for (Iter_type p = c.begin() ; p != c.end() ; ++p) // Good
		{
		…
		}		

199.		The increment expression in a for loop will perform no action other than to change a single loop parameter to the next value for the loop.		Add 6.29
200.		Null initialize or increment expressions in for loops will not be used; a while loop will be used instead.		Add 6.29
201.		Numeric variables being used within a for loop for iteration counting shall not be modified in the body of the loop.	6.29	
202.	Expressions	Floating point variables shall not be tested for exact equality or inequality.	6.4	
203.		Evaluation of expressions shall not lead to overflow/underflow (unless required algorithmically and then should be heavily documented).		
204.		A single operation with side-effects shall only be used in the following contexts: by itself the right-hand side of an assignment a condition the only argument expression with a side-effect in a function call condition of a loop switch condition single part of a chained operation. 204.1: The value of an expression shall be the same under any order of evaluation that the standard permits.	6.23 6.24	Make the reference to 204 for 6.23 read 204.1, not 204. Add 6.25
205.		The volatile keyword shall not be used unless directly interfacing with hardware.		
206.	Memory Allocation	Allocation/deallocation from/to the free store (heap) shall not occur after initialization. Note that the “placement” operator new(), although not technically dynamic memory, may only be used in low-level memory management routines. See AV Rule 70.1 for object lifetime issues associated with placement operator new().	6.40	
207.		Unencapsulated global data will be avoided.		Add 6.62
208.	Fault Handling	C++ exceptions shall not be used (i.e. throw, catch and try shall not be used.)	6.36 6.51	Add 6.37
209.	Portable Code	The basic types of int, short, long, float and double		Add 6.56

		shall not be used, but specific-length equivalents should be typedef'd accordingly for each compiler, and these type names used in the code.		
210.		Algorithms shall not make assumptions concerning how data is represented in memory (e.g. big endian vs. little endian, base class subobject ordering in derived classes, nonstatic data member ordering across access specifiers, etc.) 210.1: Algorithms shall not make assumptions concerning the order of allocation of nonstatic data members separated by an access specifier.		
211.		Algorithms shall not assume that shorts, ints, longs, floats, doubles or long doubles begin at particular addresses.		Add 6.56
212.		Underflow or overflow functioning shall not be depended on in any special way.		Add 6.56
213.		No dependence shall be placed on C++'s operator precedence rules, below arithmetic operators, in expressions.	6.23 6.24	Add 6.56
214.		Assuming that non-local static objects, in separate translation units, are initialized in a special order shall not be done.		Add 6.56
215.		Pointer arithmetic will not be used.	6.12	
216.	Efficiency Considerations	Programmers should not attempt to prematurely optimize code.		
217.	Miscellaneous	Compile-time and link-time errors should be preferred over run-time errors.		
218.		Compiler warning levels will be set in compliance with project policies.		
219.	TESTING Subtypes	All tests applied to a base class interface shall be applied to all derived class interfaces as well. If the derived class poses stronger postconditions/invariants, then the new postconditions /invariants shall be substituted in the derived class tests.		
220.	Structure	Structural coverage algorithms shall be applied against flattened classes.		
221.		Structural coverage of a class within an inheritance hierarchy containing virtual functions shall include testing every possible resolution for each set of identical polymorphic references.		Add 6.45