CS3203 Code Review Checklist

Violations fall in one of three categories [Minor/Moderate/Major]. Some violations may be in more than one category depending on the severity of the actual violation.

Rubric	Details and Examples
High-level Design	
D1: Implementation mismatch [Moderate]	• Class/UML diagrams shows class/component/method that is not actually implemented in code • Class/UML diagrams shows class/method that is named differently or with different parameters from what was implemented in code • Class/UML diagrams shows incorrect relationship between classes – E.g. UML shows inheritance while implementation shows composition
D2: Bad code organization [Moderate]	• Source files placed in incorrect directories • Inadequate organisation of source files in directories • Directory organisation structure does not reflect the documented design
D3: Breaking abstraction [Moderate/Major]	• Directly accessing class members when API should be used instead
Software Engineering Principles [Major]	
S1: DRY	• Repeating code with similar structure • Duplicate code with/without variations which can be refactored using a higher-order abstraction
S2: SRP violation	• Violating single responsibility principle • Methods/Functions that does more than one thing

Rubric	Details and Examples
S3: OCP violation	• Violating Open-close Principle: code implemented that does not allow extensibility without modifications. – Not using polymorphism – Insufficient abstractions
S4: LSP violation	• Breaks Liskov Substitution Principle: program should remain unchanged if sub type B is replaced by base type A – E.g. *mutable* Square subclass from Rectangle. Modifying one side of Rectangle does not affect the other, but modifying one side of Square implicitly modifies other side. Thus, code might break if Square was used where Rectangle is expected.
S5: ISP violation	• Interface Segregation Principle: Clients should not be forced to depend on methods they do not use – Better to have many smaller interfaces than few big ones.
S6: DI violation	• Dependency Inversion: – Higher level modules should not depend on lower level modules. Both should depend on abstractions – Abstractions should not depend on the details. Details should depend on abstractions
S7: IoC violation	• Inversion of Control: code should receive control from outside and not create it itself • Use Dependency Injection to "inject" a dependent instance into the class • "IoC and DI look similar, but they are not the same. IoC is agnostic of high-level or low-level modules and the direction of the dependencies between. One can perfectly adhere to IoC, but still violate the DI."
S8: POLA violation	• Principle of least astonishment (surprise): Component of a system should behave in a way that most users will expect it to behave.

Rubric	Details and Examples
S9: LoD violation	• Law of Demeter: Avoid invoking methods of member object returned by another method – E.g. `a.getB().getC();`
Performance (for Milestone 2 and 3) [Minor]	
P1: Inefficient code	• Code not done in an efficient manner. – Linear search when binary search can be used
P2: Inefficient data structure	• Linear search through array over using sets • Not using maps with keys, indexed arrays, etc.
P3: Redundant data storage	• Duplicating data instead of storing pointers. – E.g. When using a map to hash to elements in a vector, map should hash to pointer of element instead of a copy of the object
P4: Debugging code	• Code that should not be included from release build – E.g. check_rep, validation, assertions • Encapsulate code in directives to disable for production build
Error Handling and Testing [Minor]	
E1: Exceptions not used	• Using special return values to indicate error when it is more appropriate to throw exception
E2: Improper logging	• Debugging output is not properly filtered and logged – E.g. simply printing to std err
E3: Uncaught exception	• Exception that could be throw but never caught

Rubric	Details and Examples
E4: Resource leak	• Not deallocating memory • Using `auto_ptr` which has been deprecated • Allocator should be responsible of the deallocation
T1: Mismatched testing	• Code does not match tests documented in report
T2: Inadequate coverage	• Inadequate coverage in unit testing or system testing • Dummy used has inadequate test coverage
T3: Inappropriate stub	• Not using stubs for unit/system testing. • Inappropriate dummy data for the test being run
General Unit / Integration Testing	• amount of test cases, coverage, is the code structured to allow for easy testing? appropriate use of stubs and dummy data?

Code Smells – **These indicate potential issues either in current code or issues that may come up soon. So it is strongly suggested to address these smells. Code smells won't affect marks or grade directly.**

Code Smells [Minor] from Clean Code	
Comments	
C1: Inappropriate information	•
C2: Obsolete comment	•
C3: Redundant comment	• A comment is redundant if it describes something that adequately describes itself. – E.g. `i++; // increment`
C4: Poorly written comment	• Comment that is more confusing than the code it is describing
C5: Commented-out code	• Code commented out should be deleted. Trust your version control system
Functions [Minor/Moderate]	
F1: Too many arguments	• No more than three arguments. Should be avoid with prejudice

Rubric	Details and Examples
F2: Output arguments	• Output arguments are counterintuitive. Readers expect arguments to be inputs, not outputs. If your function must change the state of something, have it change the state of the object it is called on.
F3: Flag argument	• Boolean arguments loudly declare that the function does more than one thing (Violates SRP). They are confusing and should be eliminated.
F4: Selector argument	• Argument can be a boolean flag, enums, integers or any type that selects the behaviour of the function. Confusing if behaviour cannot be guessed from function call – E.g. what does `calculate_weekly_salary(false)` do?
F5: Dead function	• Functions that are never called should be deleted
F6: Badly named function	• Cannot tell what function does without looking at implementation or documentation – E.g. `date.add(5);` – Does this add 5 days? Weeks? Hours? Does it modify the date instance or return a new Date instance? – Compare with `date.increase_by_days(5);` and `date.days_after(5);`
Bloaters [Minor/Moderate]	

B1: Pyramid of Doom	• Indentation level more than 3 is not encouraged
	– `int method() {`
	`// level 1.`
	`Baseif (….) {`
	`// level 2.`
	`Typicalfor (…) {`
	`// level 3. Max`
	`allowedif (….) {`
	`// level 4! Not allowed!`
	• Usually violates single responsibility principle as function is doing too many things
	• Other responsibility should be handed to other function
	– `bool process_files(…) {`
	`for (string name : files) {`
	`File f = read_file(name);`

Rubric	Details and Examples

	`if (!f)`
	`  return false;`
	`return process_file(name);`

B2: Long method	• Method contains too many lines of code, i.e. more than 10 lines
B3: God class [Major]	• Class contains too many methods and fields. Tries to do everything like a god
B4: Primitive obsession	• Using primitives instead of custom type or small class – E.g. `int user_id;` – Prefer `typedef int user_id;`
General [Minor]	
G1: Overridden safeties	• Ignoring compiler warnings
G2: Dead code	• Code that is never executed – E.g.: body of if statement that is never executed
G3: Inconsistency	• Different naming conventions • Different coding styles
G4: Vertical separation	• Variables and functions should be defined close to where they are used. • Local variables should be declared just above their first usage, and have limited scope
G5: Inappropriate state [Minor/Moderate]	• Static functions/classes that should be non-static – `HourlyPayCalculator.calulate_pay(employee, overtime_rate)`
G6: Singleton pattern [Moderate/Major]	• Use of singleton pattern. Class becomes globally accessible.

Rubric	Details and Examples
G7: if-else-switch chain [Moderate/Major]	• Using switch-case or if-else in any other place besides in factory class/construction code. • Polymorphic objects should be used instead
G8: Magic numbers	• Magic numbers should be replaced by CONSTANTS • Not limited to numbers. Can be any token whose value is not self-describing – E.g. `assertEqual(7777, Employee.find("John Doe").employeeNumber());` – Intent of 7777 and "John Doe" is not clear. – Consider `assertEqual(HOURLY_EMPLOYEE_ID, Employee.find(HOURLY_EMPLOYEE_NAME).employeeNumber());`
G9: Double negatives	• Avoid `if (!buffer.shouldNotCompact())` • Prefer `if (buffer.shouldCompact())`
G10: Bad coding style	• Code that is badly written, e.g. – `if (c1 == c2) return false; else return true;` • Code that is overly complex and hard to understand

Useful References

https://refactoring.guru/refactoring/smells
https://google.github.io/styleguide/cppguide.html
http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines