This document presents a checklist for larger (object-oriented) programs, especially in the course *Programming Methods* (2IPC0).

**Requirements**

1. Understand and analyze the requirements. Preferably, precise requirements are available in a written document.

**! Coding Standard**

2. Adhere to a good coding standard for a readable layout, through systematic indentation, spacing, and empty lines. There is a (mild) coding standard for this course [1].

**Naming**

3. Use appropriate identifiers to name entities. Local entities can be designated by shorter names. Java naming conventions:
   - Class names are (singular) nouns, starting with a capital letter: `Card`
   - Method names are verbs (or begin with a verb), starting with a lower case letter: `turnCard()`
   - Variable names (including instance variables, local variables, and parameters) are nouns, starting with a lower case letter: `card`
   - Constants are written in all upper case: `QUEEN`
   - Use *camelCasing* to distinguish words in a name; except in constants, use underscores: `CardDeck, getCard(), MAXIMUM_RANK`

**Constants**

4. Avoid *magic literals*; use named constants:

```
public static final int MAXIMUM_RANK = 13;
```

**Auxiliary variables**

5. Use auxiliary variables to reduce the complexity of expressions, to avoid code duplication, to improve efficiency, and to facilitate focused comments.

**Coding idiom**

6. Use appropriate coding idiom to reveal the code's intention, in particular for selection (`?:`, **`if-else`**, **`switch-case-break`**) and repetition (**`for`**, **`while`**, **`do-while`**).

**! Procedural abstraction SRP**

7. Avoid large method bodies and (deeply) nested control structures; decompose functionality into multiple methods, through procedural abstraction. Each method must serve a well-defined purpose (Single Responsibility Principle) specified in a contract. Be aware of the pros and cons of recursive methods.

**Prefer local declarations**

8. Declare variables as locally as possible; from most preferred to least preferred: within a statement block (e.g., inside a loop body), local to a method body, as a method parameter, non-public instance variable of a class, public instance variable of a class. Use **`final`** if the value should not change.

**Method coupling**

9. Communicate data between methods via parameters and return values; minimize communication where methods refer directly to variables that are *global* to these methods.

**! Unit tests TDD**

10. Provide unit tests for key functionality. Aim for 100% branch coverage. Apply Test Driven Development (TDD): (1) specify functionality in contracts, (2) develop tests, (3) implement functionality, (4) execute tests, (5) use functionality.

**! Robustness**

11. Use `assert` statements and exceptions to signal abnormal conditions, and thus make facilities robust. Avoid the use of exceptions for normal operation (less clear control flow; run-time penalty). Check the proper throwing of exceptions in unit tests.

**! Data abstraction**
**• Enum**
**• Record**

12. Bundle related variables in a **`class`**. Avoid large classes (data decomposition).
    (a) Consider an **`enum`** to define related constants.
    (b) Consider a record (cf. *tuple*) that has only has public instance variables, when there is no concern about data representation. Optionally provide constructors that set the instance variables, and `toString()` conversion.

**• ! ADT**

    (c) Consider an Abstract Data Type (ADT) with private instance variables to provide data abstraction (hide the data representation from clients); provide public methods to access the data. See to it that methods either
    - inspect the state (also known as queries), or
    - modify the state (also known as commands),

    but not do both. Provide a class contract via public invariants between queries, and contracts for each method. For the implementation, provide a (private) representation invariant and an abstraction function. Cf. *Strategy Design Pattern*.

**Iterators**

13. Use iterators, preferably standard iterators in a for-each statement, instead of ad-hoc loops. Provide (standard) iterators. Cf. *Iterator Design Pattern*.

**Coherence**

14. Define functionality as close as possible to the data that it operates on (coherence).

**Packages**

15. Put related classes together in their own package. Explain the relationship and development status in `package-info.java`.

**Decoupling**
**DIP**

16. Avoid mutual dependencies; decouple functionality through callbacks, also knowns as listeners or observers (cf. *Observer Design Pattern*, and *Dependency Inversion Principle*).

**Composition/ Inheritance**

17. Prefer association and interfaces over inheritance.

**JCF**

18. Reuse standardized facilities, such as the Java Collections Framework.

**Design Patterns**

19. Apply common Design Patterns. See [2].

**DRY**
**SOLID**

Keep in mind: avoid code duplication (Don't Repeat Yourself); aliasing, sharing; mutable versus immutable classes; static members; inheritance, abstract classes, interfaces; mutually related classes (package level invariants); nested classes; generics; annotations; choice of algorithm and data representation; Graphical User Interface (GUI) mechanisms (event driven); the SOLID OO design principles.

## References

[1] *Coding Standard* for the Course 'Programming Methods', (2IPC0).

[2] Eddie Burris. *Programming in the Large with Design Patterns*. Pretty Print Press, 2012.