- **Obligated** features:
    - An abstract class, derived classes and polymorphism.
    - A **Model-View-Controller (MVC) pattern** for the interaction between the game-state, graphical interface in SFML and the interactive player and rules of the game. You need to use the **Observer pattern** for updating the **View** when the **Model** changes. This combination of patterns should allow you to **completely separate the visualisation from the game logic**.
    - Use one class to represent **the game**: an object that combines the different parts of the game.
    - Implement a simple **Stopwatch** helping class that keeps the time between two ticks because not all PCs run at the same speed and still all your entities should move at the same pace on all PCs. To implement this, you must use the **C++ functionality and not the SFML Clock class.** Also, provide a **Transformation** class that converts pixel values to the visible 2D game world space of **[-4,4] x [-3, 3]**. The game logic library will only use the [-4,4] x [-3,3] coordinate system and will know nothing about the actual pixel resolution of the screen. Implement these classes using the **Singleton** pattern.
    - There should be **multiple levels** in the game. These levels must be read from a file (XML, JSON, etc) using an external library. You can use the BOOST library or any other library to do so. If this external library is not available on the reference PCs (the BOOST library should be available), this must be added to the project.
    - Use **namespaces** to make modular design of your game more clear.
- Use of **smart pointers** throughout the whole project is **obligated**.
- Use **exception handling** to catch and deal with possible errors (images files not found, initialisation of graphical environment, reading of level files, etc).
- To simplify the **collision detection**, work with rectangular shapes only. Don't focus too much on collision detection. Use the easiest, but **working** solution you can implement.
- Be creative but implement at least the basic entities. **Make it work first; then make it fancy. Not** the other way around.
- For the visual implementation, the SFML library (http://www.sfml-dev.org/) must be used. You can use your own images or images you find online to represent the game entities.
- There are plenty of sources about game design patterns, do your research up front:
    - http://www.gamasutra.com/view/feature/2280/the_guerrilla_guide_to_game_code.php
    - http://www.mine-control.com/zack/patterns/gamepatterns.html
    - http://content.gpwiki.org/index.php/Observer_Pattern
    - https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller
    - https://en.wikipedia.org/wiki/Observer_pattern

- https://en.wikipedia.org/wiki/Singleton_pattern
- https://gameprogrammingpatterns.com/
- …

Practical

- A basic **working** implementation of the game with the obligated features is sufficient to get a passing grade.
- Apply proper **code commenting & documentation** of your API. This is implied to get a passing grade.
- Your code must be appropriately formatted using the **formatting rules of prog2**.
- The **CLion Code Inspection** feature will be used to assess code quality.
- Apply a **logical structure of the code & code base**, proper class construction that proves you understand C++. This is obviously implied to get a passing grade.
- Use **CMAKE** as your build system. **There must be run.sh (bash/shell) script that runs the CMAKE commands to build, install and run your game automatically.**
- Extensions are **up to you,** but put them **in your report** to be sure I notice them:
  - Multiple levels
  - Extra obstacles
  - …
- **Attention**: projects that do not compile or work (e.g., compiling error or segmentation faults when starting the application) automatically imply "student **failed** this part". The computers in G026 are used as the reference platform: **your code should compile and work on those computers**.
- The project has to be made and handed in **individually**. Of course, you can discuss design/problems/solutions with other students as much as you like. Projects will be **tested on plagiarism**. If tested positive, the grade will be zero or worse.
- Describe your design and some of the choices you made in **a report** (ca. 2 A4 pages). This report has to reflect that you really thought about the choices you made for your design. Additionally, provide a **README** which explains the rules and controls of your game.
- Implement features **incrementally**! Make a **private Github repository** (using your student Github account) to **frequently commit your code** increments and **enable Travis** for your project. **Invite gdaneels for this Github repository.** The final version of your project must show a successful build on Travis.
- Good luck! If you have any questions, remarks or comments: glenn.daneels@uantwerpen.be or ask me during the lab sessions.
- Deadline: To be announced. (Somewhere in **January 2020**)
- Submit the final project and report **on Blackboard and by mail and by Github.**