Systems Engineering and Data Processing in C++

Summer Term 2021

Hasso Plattner Institute
Computer Graphics Systems Group

Prof. Dr. Jürgen Döllner
and Willy Scheibel

Assignments #4: Class Design and Class Hierarchies

Objectives

In these assignments you will learn how to:

- design and implement user-defined types, domain-specific interfaces, and implementations as C++ classes,
- design and implement class hierarchies to provide parameterizable program behavior with polymorphic objects, and
- different representations of class relationships and patterns in C++.

Rewards

For each successfully solved task you and your partner (optional) gain one bonus exam point. A serious shortcoming in fulfilling the requirements of a task will result in no bonus point regarding this task. You can gain at most 4 bonus points through this exercise sheet.

Submission

All described artifacts has to be submitted to the course moodle system[1] as a zipped archive. The naming convention includes the assignment number and your *personal assessment numbers* with following naming convention: `assignment_4_paNr1.zip` or `assignment_4_paNr1_paNr2.zip` whether or not pair programming was applied. Compiled, intermediate, or temporary files should not be included. If pair programming is used, the results are turned in only once. The assignments #4 are due to the next tutorial on June 17th, 9:15 a.m. GMT+2.

Presentation

Feel free to present any of your solutions during the Zoom meeting in the upcoming tutorial. If you want to present your solution of a task, please send the source code beforehand via e-mail to willy.scheibel@hpi.de.

General Instructions

Pair Programming On these assignments, you are encouraged (not required) to work with a partner, provided you practice pair programming. Pair programming "is a practice in which two programmers work side-by-side at one computer, continuously collaborating on the same design, algorithm, code, or test." One partner is driving (designing and typing the code) while the other is navigating (reviewing the work, identifying bugs, and asking questions). The two partners switch roles every 30–40 minutes, and on demand, brainstorm.

Cross-Platform The assignments can be solved on all major platforms (i.e., Windows, Linux, macOS). The evaluation of submitted assignments can be carried out on any of these platforms. All results should not use platform-specific dependencies. Platform-specific build and macro code is permitted.

Violation of Rules a violation of rules results in grading the affected assignments with 0 points.

- Writing code with a partner without following the pair programming instructions listed above represents a serious violation of the course collaboration policy.
- Plagiarism represents a serious violation of the course policy.

[1] https://moodle.hpi.de/course/view.php?id=174

Task 4.1: Dice Games

Description: Implement two variants of a dice game based on one single C++ framework. Design a framework that represents the key application concepts *game*, *player*, and *dice*. The framework should allow for simulating n players.

Variants:

- "7 counts!" — A player always rolls the two dice at the same time. If the sum of the dice is seven, then his or her score gets incremented. Several players form a team. All team players can roll the dice a fixed number of times. The winner is the person with the highest score.
- "Stuck-in-the-mud!" — A variant of the game uses five dices. Players can only score on a roll which does not include the numbers 2 and 5. Any dice, which show a 2 or a 5, become "Stuck in the mud!". If numbers 2 and 5 are not included, the total increments a player's score.

Implement one command line program for each of the two variants that simulates a game. The two programs should take two positional command line arguments: (1) the number of players and (2) the number of simulation rounds. After performing the simulation, the resulting scores should be printed to the console in the form "`<player_num>: <score>\n`". Name the source files containing the entry points for the two programs `dice/seven-main.cpp` and `dice/mud-main.cpp`, respectively.

Provide a lean class design and implementation, i.e., stay focused on the game's core requirements and do not "overengineer" your classes, e.g., by anticipating future or devised needs. The compactness and the comprehensibility matters.

Artifacts:

a) `dice/seven-main.cpp`, `dice/mud-main.cpp`, `dice/...` : Source code of the solution.

Objectives: OOP, abstract class design

Task 4.2: Caching in a Bezier Curve Computation Class

Files: `beziercurve/test.cpp`, `beziercurve/CubicBezierCurve.h`

Description: Design and implement a concrete type that implements the concept of a cubic Bézier curve C. A Bézier curve is a parameterized curve $\{ C(t) \mid t \in [0,1] \}$ that interpolates its end points and approximates its inner control points.

- C is defined by 4 control points $\{p_0, ..., p_3\}$.
- A control point p is represented by a three-component vector of doubles, i.e., $p = (x, y, z) \in \mathbb{R}^3$.
- The end control points are interpolated, i.e., $C(0) = p_0$ and $C(1) = p_1$.
- The curve points $C(t)$ are computed based on a weighted sum of the control points; weights are defined by cubic Bernstein polynoms:

$$C(t) = \sum_{i=0}^{3} \binom{3}{i} (1-t)^{3-i} t^i p_i$$

If no control points are given, they are set to $(0, 0, 0)$. The control points can be changed by the application at any time. The parameter interval $[0, 1]$ represents a geometric parameterization. The class overloads the `operator ()`. This way, a Bézier curve object can be used like a function. To speed up computation, add a kind of cache based on a discrete pre-computed value table; the caching should not have any impact on the way a curve object is used. Do not extend or parameterize the class, e.g., with respect to the curve degree or the data type of control points. Concentrate on a small, lean interface and a robust, efficient implementation (e.g., de Casteljau's algorithm). In particular, do define appropriate constructors (e.g., argument lists, copy/move, ...) and take care of the internal cache.

Adapt the marked locations in the test case to use your class interface, if necessary.

Artifacts:

a) `beziercurve/test.cpp`, `beziercurve/CubicBezierCurve.h`, `beziercurve/...` : Source code of the solution.

b) `beziercurve/c1.txt`, `beziercurve/c2.txt` : Test results.

Objectives: OOP, class design, class state handling

Competition: This task can be extended with respect to dynamic memory usage and compile-time computation, i.e., an excelling implementation uses no dynamic memory and computes the final state of the program during compilation. If you want to compete in the open competition submit your isolated source code of this task via e-mail to willy.scheibel@hpi.de until June 15$^{\text{th}}$. Your source code is checked for conformity with respect to the expected output. If the source code behaves as expected, we assess the compile-time code share and measure dynamic memory usage.

Task 4.3: Design of a Simple Graph Class Hierarchy

Files: `graph/Graph.h`, `graph/main.cpp`

Description: Design and implement a small, simple framework for *directed acyclic graphs* (DAG). Each node can have zero, one or more child nodes, depending on its subtype. No cycles are allowed. Each node defines a weight. Leaf nodes specify a concrete weight, while group nodes sum up the weights of their child nodes. A special node class, the `Proxy`, is used to virtually include subgraphs – in this case, the proxy's weight is the subgraph's weight.

Design and implement an object-oriented solution using inheritance. The required classes are:

- **`Node`**: Abstract base class for all kinds of node objects.
- **`Group`**: A container node having zero or more child nodes.
- **`Leaf`**: A leaf node, specifying a weight.
- **`Proxy`**: A node delegating to another subgraph.

Two core functionalities should be implemented:

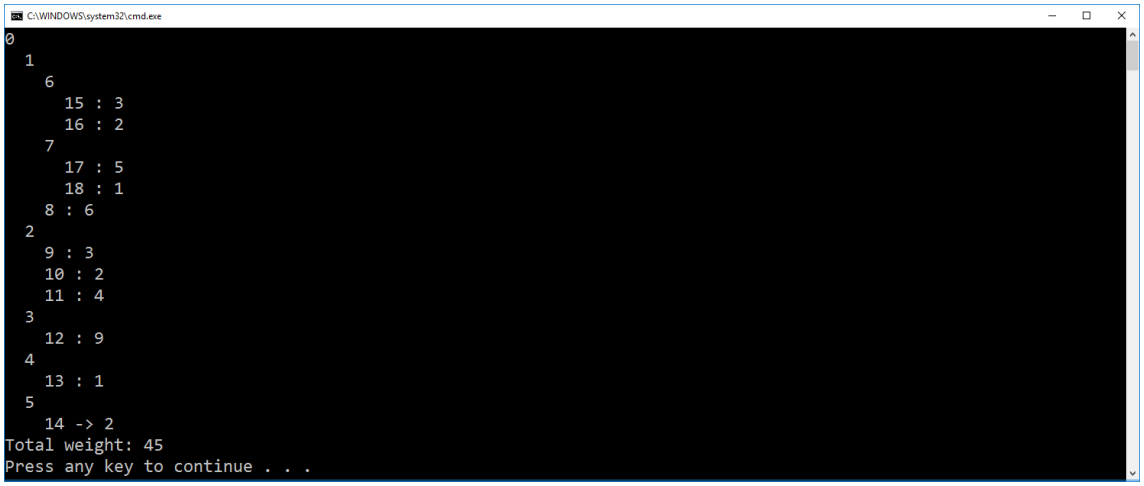
- *Graph printing* by means of line outputs (use indentation).
- *Total weight computation* for an entire graph.

Graph Data Specification

- A Leaf Node $\{L, i, w\}$, is specified by its identifier i and its weight w.
- A Group Node $\{G, i, j_1, ..., j_m\}$, is specified by its identifier i, the number of child nodes m and their identifiers j.
- A Proxy Node $\{P, i, k\}$, is specified by its identifier i and the referring subgraph k.

Use virtual functions defined in an abstract base class for all type of nodes. Do not extend the functionality beyond the small features described here to keep it simple. The nodes should be allocated dynamically. The graph is built by mapping the given input to a graph structure; it is not modified afterwards. For the given test data, produce the line-based output and compute the total weight.

Example:



Artifacts:

a) `graph/Graph.h`, `graph/...` : Source code of the solution.

Objectives: abstract classes, subtyping, inheritance, polymorphism, late binding

Task 4.4: Multiple Inheritance and Alternatives

Files: `transactions/Transaction.h`, `transactions/Transaction.cpp`, `transactions/main.cpp`

Description: Assuming there is an domain-specific transaction class named `Transaction`, i.e., a high-level class that processes business operations. For simplicity, this key function shall be implemented by the method `void process(string)`.

The OOA group comes up with a proposal to extend that concept regarding transaction logging (`LoggedTransaction`) and secured transactions (`SecuredTransaction`). As different kinds of transactions modify the processing, the base class `Transaction` defines two protected virtual functions `void onBeforeProcess(string)` and `void onAfterProcess(string)`, that are supposed to implement the specific variations in each case. Both, logged transactions as well as secured transactions implement their behavior by means of these two protected functions.

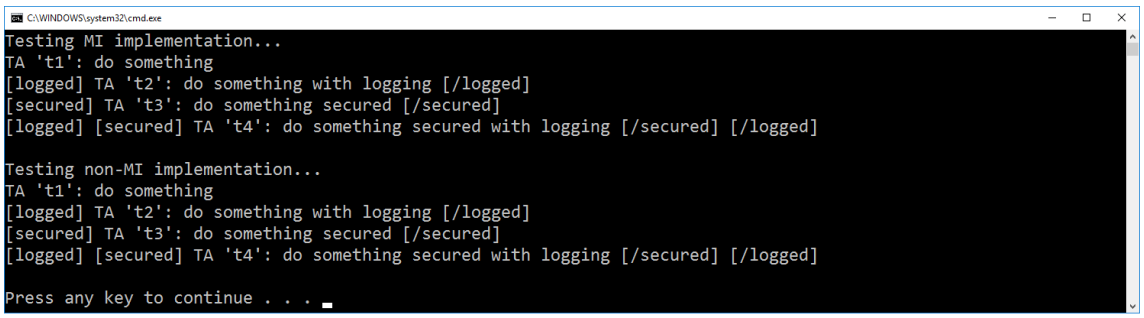
Most application cases, however, will demand for both, logged and secured transactions.

a) Provide an implementation based on multiple inheritance, i.e., `LoggedSecureTransaction`.

b) Provide an alternative object-oriented design that does not use multiple inheritance to model and implement transaction classes that can optionally be logged and/or secured.

- Use namespaces to separate both implementations.
- Adapt the given test case to use your solutions.
- You are not required to provide an actual implementation of logging or securing beyond the command line output shown below.

Example:



Artifacts:

a) `transactions/Transaction.h`, `transactions/Transaction.cpp`, `transactions/main.cpp`, `transactions/...` : Source code of the solution.

Objectives: class hierarchies, [avoiding] multiple inheritance