# Problem Set 6

Due before midnight on Wednesday, November 9, 2016.

## 1 Assignment Goals

1. Learn how to use protected derivation to increase modularity and separation of function.
2. Practice using delegation derivation to produce a tight interfaces.
3. Learn how to pass parameters to a base class constructor.

## 2 Problem

The game of Kalaha is described in problem set 5. In that problem set, you implemented three types: `Player`, `BoardFrame`, and `PitFrame`. Together, they represent the structure of a Kalaha board, but they make no mention of the seeds used to play the game.

Seeds are moved from pit to pit during the play of the game. In this problem, you will extend your previous solution to include the seeds. There are many ways to do this. The simplest (but less modular) way might be to add an integer data member to each pit to record the number of seeds it contains, to modify the board and pit constructors to take seeds into account, and to add more member functions to both the board and pit classes to manipulate seeds.

However, there is a logical difference between the *structure* of the board and the *state* of an in-play Kalaha game. PS5 focused just on the structure. Here, we want to keep the state isolated from the structure as much as possible. The structure is constant during a play of the game, whereas the state changes on each move.

To maintain this separation, you will augment your previous solution by adding two new classes, `Pit` and `Board`. `Pit` should be derived from `PitFrame` using `public` derivation, and `Board` should be derived from `BoardFrame` using `protected` derivation.

Class `Pit` will have an additional private integer data member storing the number of seeds in the pit. It's constructor takes the same parameters as the `PitFrame` constructor and passes them on to its base class via a ctor.

Class `Board` will have a constructor that takes two arguments, the number of houses and the initial number of seeds. It uses a ctor to construct the board frame. Then, in the body of the constructor, it initializes the number of seeds in each pit. It will have other member functions that provide functionality that will be needed later to actually play the game.

More specifically, class `Pit` should implement several sets of public functions in addition to its constructor and destructor.

1. Delegates from PitFrame.

```
Pit* next() const;
Pit* getOpposite();
```

The counterparts of these functions in `PitFrame` return `PitFrame` pointers, not `Pit` pointers, so the return values will need to be cast appropriately.

2. Inline helper functions.

   - `bool isStore() const {...}`
     returns `true` if this pit is a *store* (as opposed to a *house*).
   - `bool belongsTo( Player pl ) const {...}`
     returns `true` if this pit is owned by player `pl`.
   - `bool isEmpty() const {...}`
     returns `true` if there are no seeds in the pit.
   - `int getSeeds() const {...}`
     returns the number of seeds in the pit.
   - `void drop() {...}`
     drops a seed into the pit.
   - `void collect(int n) {...}`
     drops `n` seeds into the pit.
   - `int grab() {...}`
     removes the seeds from a pit and returns their number, leaving the pit empty.

3. Out-of-line print function.

   ```
   ostream& print(ostream& out) const;
   ```

   prints the number of seeds in a width 2 fields enclosed in square brackets, e.g., "`[ 4]`".

Class `Board` should implement several public functions in addition to its constructor and destructor.

1. `Pit* getPit(Player pl, int pitNum) const {...}`

   returns a pointer to `Pit` number `pitNum` owned by player `pl`.

2. `Pit* myStore(Player pl) const;`

   returns a pointer to the *store* of player `pl`.

3. `int houseSeeds(Player pl) const;`

   counts the total number of seeds in all of player `pl`'s *houses* (but not including her *store*).

4. `Pit* sow(Player pl, int startPit);`

   performs the *sow* operation for player `pl` starting in pit number `startPitNo`, as described in handout 6.

5. `void capture(Player pl, Pit* endPit);`

   performs the *capture* operation for player `pl` whose sowing ends in pit `endPit`. Recall that a capture takes place if the last seed was dropped into an empty house owned by the player, and the opposite house (owned by the opponent) is non-empty.

6. `int score(Player pl) const;`

   returns player's `pl` *score*, which is the number of seeds in its *store*.

7. `ostream& print(ostream& out) const;`

   Prints the board as shown in Figure 2 of handout 6.

## 3 Testing

You should modify your test program from PS5 to take a second command line argument, `numSeeds`, that gives the initial number of seeds to place in each house. It will have to pass this argument to the board constructor to be made available in one way or another to the functions that need that information. It should then test each of the functions in the two new classes `Pit` and `Board`.

A test consists of preparing and printing the arguments, calling the function, and printing the result. Note that to test certain functions, you may need to prepare the board by setting its state to a particular state that is different from its initial state. Also, more than one test is often required to adequately test a function that takes conditional actions.

There are now two ways to print a pit: `Pit::print()` only prints the number of seeds; `PitFrame::print()` prints the owner, the pit number, the label, the label of the next pit, and the label of opposite pit, but *not* the number of seeds. Both of these functions may prove useful in presenting the results of your tests.

## 4 Programming Notes

In an ideal world, you would not need to make any changes at all to the `PitFrame` and `BoardFrame` classes. However, you will likely need to make a few minor changes to `BoardFrame.hpp` and `BoardFrame.cpp` in order to get your code to work.

1. The `BoardFrame` constructor from PS5 constructs a board consisting of several `PitFrame` objects. For this problem set, it needs to know to construct `Pit` objects instead since `PitFrame` objects, since the latter do not contain storage for the `seeds`. To accomplish this, you may directly modify the `BoardFrame` constructor to create `Pit` objects instead of `PitFrame` objects.[1]

2. Since `BoardFrame` doesn't know anything about seeds (and I don't want it to since seeds have nothing to do with the structure of the board), the `seeds` data member of each `Pit` will have to be set by the `Board` constructor *after* the pits have been created. In order to avoid a privacy violation, you should make `Board` a friend class of `Pit` just as you made `BoardFrame` a friend class of `PitFrame` in PS5. Do not use a public "setter" function for this purpose since setter functions are not restricted to just the one class.

3. `BoardFrame.hpp` will need to `#include Pit.hpp`.

If you feel you need to make any other changes to either `BoardFrame` or `PitFrame`, please email me with a complete copy of your project and I can perhaps suggest ways to avoid those changes. In any case, write a list of any such changes and your reasons for making them, and submit that list with your solution.

[1] This of course would not be possible if `BoardFrame` were in a library that you did not control.

# 5 Grading Rubric

Your assignment will be graded according to the scale given in Figure 1 (see below).

| # | Pts. | Item |
|---|---|---|
| 1. | 1 | A well-formed `Makefile` or `makefile` is submitted that specifies compiler options `-O1 -g -Wall -std=c++11`. |
| 2. | 1 | Running `make` successfully compiles and links the project and results in an executable file `test2Kalaha`. |
| 3. | 6 | `test2Kalaha` adequately tests all functions in the new classes as required in section 3. |
| 4. | 4 | `test2Kalaha` gives correct answers to all tests when run with arguments 3 2 and with arguments 6 4. |
| 5. | 4 | `BoardFrame` and `PitFrame` are unchanged from PS5 except for necessary modifications as described in 4. All such changes are documented. |
| 6. | 3 | All of the instructions in sections 2 and 4 are followed. |
| 7. | 1 | All relevant standards from previous problem sets are followed regarding good coding style, submission, identification of authorship on all files, and so forth. |
| | 20 | Total points. |

Figure 1: Grading rubric.