Code review – Exercise 2

Resharper!

We urge you to use resharper!
So many of your mistakes could have been avoided if you have used it. It will help you write better code and learn much about C++ in the process.

While reviewing your code in exercise 2, we have encountered some code errors, misuses, bad practices and design issues. There was a big improvement from exercise 1, but still there were a few things we wanted to get to your attention.
The purpose of this document is to present the code issues that we have found to help you avoid these kind of errors in the following submissions. All the code snippets here are taken directly from your work, so don't be surprised if they look somewhat familiar. It might be from your exercise. Not all examples shown here caused point reduction, but we wanted you to learn from them for better C++ coding practices.

Do's and Don'ts – Ex2

C-style cast used instead of a C++ cast:

We decided not to take off point on this one, this time, but we will next time.
It is important to understand which cast you are using. It will help you avoid bugs.

C-style: `(const char**)board`

C++-style: `const_cast<const char**>(board)`

C-style: `(GetAlgoFuncType)GetProcAddress(player.hDll, "GetAlgorithm");`

C++-style: `reinterpret_cast<GetAlgoFuncType>(GetProcAddress(player.hDll, "GetAlgorithm"));`

Assigned values are never used:

Multiple lines:

```
isVertical = findEnemyAttackedLocationInGivenDirection(row, col, 1, 0, curSunkShip);
isVertical = findEnemyAttackedLocationInGivenDirection(row, col, -1, 0, curSunkShip);
```

Testing the wrong return value:

```
fileStatus = FindNextFileA(hFind, &FindFileData);
if (hFind == INVALID_HANDLE_VALUE)
{
	status = STATUS_ERROR;
	addErrorMsg(MISSING_ATTACK_IDX, MISSING_ATTACK_MSG);
}
```

Hiding a member function of a super class:

In Super Class:

```
void Player::setSide(bool sideA) {
	isPlayerA = sideA;
}
```

In Derived Class:

```
class FileReaderAlgo : public Player {
public:
void setSide(bool sideA) {
		isPlayerA = sideA;
		isPlayerB = !sideA;
	}
}
```

You should either use a different name, or make the super method virtual, and then use the override classifier in the derived class.

Like so:

In Super Class:

```
virtual void setSide(bool sideA) {
	isPlayerA = sideA;
}
```

In Derived Class:

```
class FileReaderAlgo : public Player {
public:
void setSide(bool sideA) override {
		isPlayerA = sideA;
		isPlayerB = !sideA;
	}
}
```

Overriding functions without override specifier

```
class FileAlgo : public IBattleshipGameAlgo {
public:
	bool init(const std::string& path);
	//...
};
```

needs to be:

```
class FileAlgo : public IBattleshipGameAlgo {
public:
	bool init(const std::string& path) override;
	//...
};
```

We decided not to take off point on this one, this time, but we will next time.

It is good practice to declare when overriding a method, for several reasons:
1. The compiler enforces your declaration, so if you made a mistake in the methods' signature, it will tell you and help you avoid bugs.
2. If some other developer looks at your code, he knows that this method is a derived method which allows him to better understand the code.

Inconsistent linkage issues:

In BasePlayer.h:

```
#define ALGO_API __declspec(dllexport)

class ALGO_API BasePlayer : public IBattleshipGameAlgo { ... }

ALGO_API IBattleshipGameAlgo* GetAlgorithm();
```

In the preceding example there three thing wrong:

1. **Re-defining ALGO_API when in is already been defined for you in IBattleshipGameAlgo.h.**
2. **Always using __declspec(dllexport) and not as seen in class. This causes inconsistent dll linkage since it shadows the declaration in IBattleshipGameAlgo.h, and the game manager includes IBattleshipGameAlgo.h expecting to dllimport GetAlgorithm().**
3. **Using __declspec(dllexport) on a class. This is a BIG NO NO! We do not export C++ entities! There is no need to export a class that is not exposed outside of our dll. The only method that is being called by the GameManager is GetAlgorithm(), hence it is the only one that needs to be dllexport-ed.**

Using default constructor – causing uninitialized parameters:

```
class FileAlgo : public IBattleshipGameAlgo {
	int playerID;
	int numRows;
	int numCols;
	std::queue<std::pair<int, int>> attackQueue;
	bool initAttackQueue(std::ifstream& attackFile);
public:
	bool init(const std::string& path) override;
};
```

A better practice:

```
class FileAlgo : public IBattleshipGameAlgo {
	int playerID;
	int numRows;
	int numCols;
	bool initAttackQueue(std::ifstream& attackFile);
public:
	BattleBoard() : playerID(0), numRows(0) , numCols(0) {}
	bool init(const std::string& path) override;
};
```

Uninitialized parameters are bad practice. We have seen it throughout the exercise as some of the students did not initialize full arrays, resulting in junk values printing.

Declarator is never used:

Unused variables often indicate a bug.

Example 1:

```
bool isInvalidLine = false;//if invalid dont add to vector of attack
```

In this case, the developer clearly was intending a use for this variable but never used it.

Example 2:

```
if (m_mode == AttackMode::RandomMode)
{
	pair<int, int> att = AllignCord(GetValidRandomAttack());
}
```

Same goes here, clearly some logical bug is bound to occur.

Function returns by const value:

There is not much sense in returning a `const` value, since the value was created in the purpose of being returned to the caller. Do not use `const` identifier in these cases.

```
const SquarePeek BoardGame::peekSquare(const BoardGamePoint& point, bool hit) const
{
	// Some Code
	// . . .

	return SquarePeek(peek, info.boardShip, lastState);
}
```

Missing header guard + Wrong header include policy:

```
#include "AlgorithmFromFile.h"
#include "NaiveAlgorithm.h"
#include "PredictiveAlgorithm.h"
#include "BoardShips.h"
#include "BattleshipPlayerStats.h"

class GameManager{ . . . };
```

1. **Especially when using many includes in a header, a header guard is important to prevent code ambiguity, compiling errors, warnings and even runtime errors! just add: `#pragma once` at the top of the header file**
2. **You should try to use as little as possible includes in .h files. In this case, there is no need for:**
   ```
   #include "AlgorithmFromFile.h"
   #include "NaiveAlgorithm.h"
   ```
   Since GameManager doesn't need to know the specific implementations of the players. Consider this: we run your player with our game manager, surely we don't include these two files and the code still works!
3. **Move as many headers to the .cpp files to shorten compilation times and to prevent include loops.**

Empty implementation of copy constructors and assignmenets:

```
Game& operator=(const Game&) {}
FixedPlayer(const FixedPlayer&) {}
FixedPlayer& operator=(const FixedPlayer&) {}
```

1. **This is a real problem. You say you have a method that does something and it does something else.**
2. **The bigger issue is since you do not declare these methods to be `explicit`, they could be called without you intended them to, for example when pushing an instance into a vector, and it will be copied.**
3. **A non-void function with no return statement.**

Polymorphic class with non-virtual public destructor:

When you use polymorphism, it is important to explicitly declare a virtual destructor so when the derived object is being destroyed, it knows to call his super's destructor as well, thus preventing memory leaks.

What not to do

```
class FilesLister {
public:
	FilesLister(const std::string & basePath);
	virtual void refresh();
};
```

What to do (Exactly what we did in IBattleShipGameAlgo.h)

```
class FilesLister {
public:
	virtual ~FilesLister() = default;
	FilesLister(const std::string & basePath);
	virtual void refresh();
};
```

Throw expression can be replaced with a re-throw expression:

The code is this:

```
catch (exception e) {
	throw e;
}
```

1. **First you have to think if this is really necessary. Catching and re-throwing an exception without any actions in redundant.**
2. **If you do decide to re-throw, use this syntax instead:**

```
catch (exception e) {
	throw;
}
```

Unreachable code:

Example 1:

```
switch (t)
{
case static_cast<char>(Symbol::ABoat):    case static_cast<char>(Symbol::BBoat):
	return len == ShipLen::BoatLen;
	break; // unreachable
}
```

Example 2:

```
void Algo::printBoard(int**& board, int numRows, int numCols){
	return;
	for (int i = 0; i < numRows; i++) { // unreachable
		printf("|");
		for (int j = 0; j < numCols; j++)
			printf("%d|", board[i][j]);
		printf("\n");
	}
}
```

Example 3:

```
else {
	/* could not open directory*/
	throw;
	cout << "Error: could not open directory << endl; // unreachable
	return;
}
```

Deleting a static allocation

```
Point potentialMoves[BOARD_SIZE * BOARD_SIZE];

RandPool::~RandPool()
{
	delete[] potentialMoves;
}
```

Missing a user defined constructor because of uninitialized data members (Column AH in grades excel)

```
class Player : public IBattleshipGameAlgo {
protected:
	int numOfRows;
	int numOfCols;
	const pair<int, int> InvalidAttack = make_pair(-1, -1);

	virtual void setBoard(int player, const char** board, int numRows, int numCols)
		{ numOfRows = numRows; numOfCols = numCols; }
};
```

If you choose to hold members in your class, you must initialize them. If you don't, you could end up with uninitialized values in some compilers, which results in junk values.

You have 2 choices:

1. Constructor:

```
public:
	Player(int num_of_rows, int num_of_cols) : numOfRows(num_of_rows),
	                                           numOfCols(num_of_cols) {}
```

2. Staticly:

```
	int numOfRows = 0;
	int numOfCols = 0;
```

OOP Light Issues:

```
class NaiveAlgo : public IBattleshipGameAlgo {
	static std::pair<int, int> up(const std::pair<int, int>& pair);
	static std::pair<int, int> down(const std::pair<int, int>& pair);
	static std::pair<int, int> left(const std::pair<int, int>& pair);
	static std::pair<int, int> right(const std::pair<int, int>& pair);
}

class SmartAlgo : public IBattleshipGameAlgo {

	static std::pair<int, int> up(const std::pair<int, int>& pair);
	static std::pair<int, int> down(const std::pair<int, int>& pair);
	static std::pair<int, int> left(const std::pair<int, int>& pair);
	static std::pair<int, int> right(const std::pair<int, int>& pair);
}
```

In our case, deciding to inherit straight from `IBattleshipGameAlgo` was probably not the best idea, since there is plenty of code that is relevant to all players, what results in code duplication. One option here was to have an Abstract Class `BaseAlgo` that inherits from `IBattleshipGameAlgo` and implements all the common code. Then the actual players could inherit from it and avoid code duplication.

OOP Medium Issues:

A good example for Medium OOP issues is a really long GameManager method which does it all:

- **Parses the input board and dlls**
- **Checks input's validity**
- **Manages the game**
- **Prints the game board**
- **Prints Errors**

In addition, it does so with no regard to Object Oriented design, meaning there is much code duplication, use of conditions and switch-case instead of generalizing the code and using a distinct class hierarchy.

OOP Bad Issues:

Luckily, we did not encounter Issues of that kind. Examples:

- **Writing the entire code in one class**
- **Not using classes at all, and running the entire game in main()**
- **Wrong understanding of the concepts of: Inheritance, Polymorphism, Abstraction, Encapsulation**

Additional Comments:

1. **Avoid having function declared but not implemented**
2. **Still some of you have really long functions! They usually indicate that a better design could be found.**
3. **Many unused directive (#includes) – we did not take off points this time.**

- **As you finish writing your code, try going to each file and ask yourself if you really need each and every #include there**
- **Move all possible #include statements to the .cpp file. If you only need the include in the .h file for a pointer/reference type you could use a forward decleration:**

```
#include "IBattleShipGameAlgo.h"

class AttackResult //This is the forward decleration

class GameUtils{
public:
	std::string res2string(AttackResult & res);
 };
```