Practice 3: Aircraft Carrier Battle

Programming 2

Academic Year 2024-2025

In this practical assignment, we simulate a battle between aircraft carriers by modeling the behavior of fighters and carriers. This assignment applies object-oriented programming concepts and UML design principles to create a dynamic simulation that reflects real-world challenges.

Submission Guidelines

- The deadline for submission of this assignment is **Friday, May 9th**, by **23:59**.
- This assignment consists of various files: `Coordinate.cc`, `Coordinate.h`, `Fighter.cc`, `Fighter.h`, `AircraftCarrier.cc`, `AircraftCarrier.h` `Board.cc` and `Board.h`. All these files must be compressed in a file called `prog2p3.tgz` that will be submitted using the assignment server as in the previous practices. To create the compressed file, you should use the following command (in a single line):

Terminal

```
$ tar cvfz prog2p3.tgz Coordinate.cc Coordinate.h Fighter.cc Fighter.h
            AircraftCarrier.cc AircraftCarrier.h Board.cc Board.h
```

Honor Code

If plagiarism (total or partial) is detected in your Assignment, you will get a **0** for the submission, and the Polytechnic School (EPS) administration will be notified for disciplinary action.

Discussing possible solutions with your classmates is acceptable.
Enrolling in an academy to assist you study and complete the assignments is acceptable.

Copying code from other students or asking ChatGPT to complete the assignment for you is not acceptable.
Enrolling in an academy so they will implement your assignments is unacceptable.

If you need help, contact your professor.
Do not copy.

General Rules

- You must submit your code exclusively through the assignment submission server of the Department of Languages and Computing Systems (DLSI). It can be accessed in two ways:

  - DLSI Main Page (`https://www.dlsi.ua.es`), option “ENTREGA DE PRÁCTICAS”.
  - Directly at `https://pracdlsi.dlsi.ua.es`.
- Important submission details:
  - The username and password for submission are the same as in UACloud.
  - You can submit the assignment multiple times, but only the last submission will be evaluated.
  - Submissions via email or UACloud will not be accepted.
  - Late submissions will not be accepted.
- Your assignment must compile without errors using the C++ compiler in the Linux distribution of the labs.
- If your assignment does not compile, the grade will be 0.
- At the beginning of all submitted source files, you must include a comment with your NIF (or equivalent) and name. For example:

prac1.cc

```
// DNI 12345678X GARCIA GARCIA, JUAN MANUEL
...
```

- Passing **all** the autograder tests qualifies you for the assignments exam. If any test fails, you will not be able to take the exam, and your grade for this assignment will be 0.
- The calculation of this assignment grade and its relevance in the final course grade are detailed in the course introduction slides (*Unit 0*).

1 Assignment Description

In this assignment, several classes will be implemented to simulate a battle between two aircraft carriers on a board representing the conflict zone. The aircraft carriers will send their fighter planes to battle for each position on the board.

2 Implementation Details

Several files necessary for the proper completion of the assignment will be posted on *Moodle*:

- `prac3.cc`. This file contains the `main` of the assignment. It is responsible for creating the objects of the classes involved in the problem and simulating a battle. This file should not be included in the final submission; it is only a tool to test the assignment and can be modified or used as a basis for other tests.
- `makefile`. This file allows for the optimal compilation of all the source files of the assignment to generate a single executable.
- `autocorrector-prac3.tgz`. This archive contains the autograder files to test the assignment with several unit tests to verify each method separately. The automatic grading of the assignment, after submission, will be performed using these same tests, and all of them must be passed in order to be eligible for the assignment exam.

In this assignment, each class will be implemented in a different module, meaning we will have two files for each one: `Coordinate.h` and `Coordinate.cc` for the board coordinates, `Fighter.h` and `Fighter.cc` to manage the fighters, `AircraftCarrier.h` and `AircraftCarrier.cc` for the aircraft carriers, and `Board.h` and `Board.cc` for the board. These files, along with `prac3.cc`, must be compiled together to generate a single executable. One way to do this is as follows:

Terminal

```
$ g++ Coordinate.cc Fighter.cc AircraftCarrier.cc Board.cc prac3.cc -o prac3
```

This solution is not optimal, as it recompiles all the source code when perhaps only some of the files have been modified. A more efficient way to compile code spread across multiple source files is by using the `make` tool. You should copy the provided `makefile` from Moodle into the directory where the source files are located and then execute the following command:

Terminal

```
$ make
```

- You can refer to slide 61 and onwards of *Unit 5* if you need more information about how `make` works.

2.1 Exceptions

Some of the methods you will create in this assignment must throw exceptions, which are a particularly useful mechanism in constructors because they prevent objects from being constructed with incorrect values.

To throw an exception, you must use `throw` followed by the type of exception. In C++ they can have any value (an integer, a string, etc.), but in this practice, you must always throw the standard exception `invalid_argument` (which should never be caught):

Terminal

```
if (...)
 throw invalid_argument(message);
```

where `message` is a `string` or a vector of `char`; if the invalid argument is a number, it can be converted to a string with the function `to_string`:

Terminal

```
if (...)
 throw invalid_argument(to_string(number));
```

To use `invalid_argument` you must include `#include <stdexcept>` in your code.

2.2 Generating Random Numbers

For the simulation of the battle between two fighters, it is necessary to generate a random number using this function (which you must include at the beginning of the file `Fighter.cc`):

Terminal

```
#include <cstdlib> // For rand()

// returns a random number between 0 and maxrnd-1
int getRandomNumber(unsigned int maxrnd){
  return rand() % maxrnd;
}
```

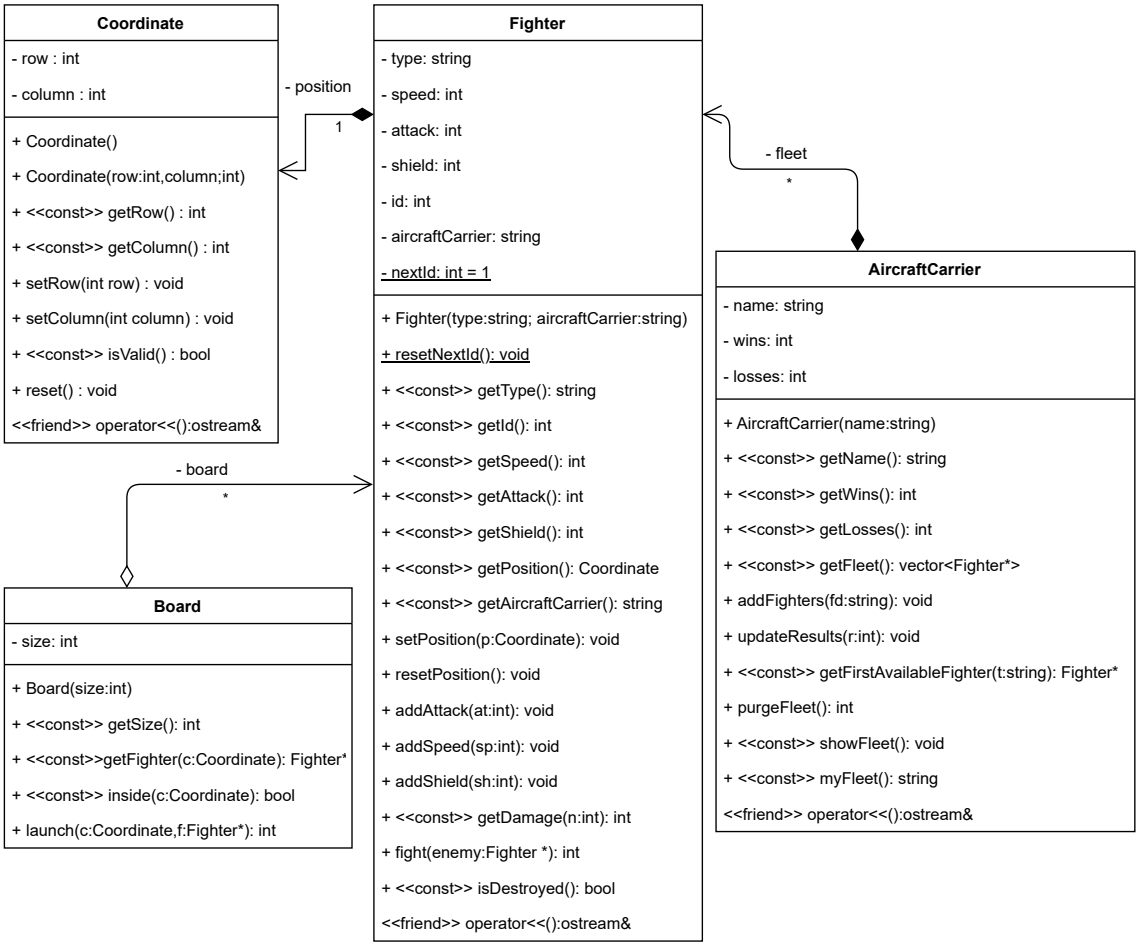
- In the file `prac3.cc`, the random number generator is initialized by calling the function `srand`. You should not modify that call.

3 Classes and Methods

The figure on the following page shows a UML diagram with the classes to be implemented, along with the attributes, methods, and relationships present in this assignment.

It is not allowed to add any public attributes or methods to the classes defined in the diagram, nor to add or change the arguments of the methods. If you need to incorporate more methods and attributes into the classes described in the diagram, you may do so, but always include them in the private section of the classes. Also, remember that the *aggregation* and *composition* relationships result in new attributes when translating the UML diagram into code. Refer to slides 58 and 59 of *Unit 5* if you have any doubts about how to translate the *aggregation* and *composition* relationships into code.

The methods of each class are described below. It is possible that some of these methods may not be necessary for the assignment, but they will be used in the autograder unit tests. It is recommended that the classes be implemented in the order in which they appear in this statement.



3.1 Coordinate

This class manages the coordinates on the board. As shown in the diagram, it has two private instance variables, "`row`" and "`column`", which represent the row and the column on the board, respectively.

The methods of this class are as follows:

- `Coordinate()`. Creates a coordinate by assigning -1 to both the row and the column. This coordinate would be an invalid coordinate.
- `Coordinate(int row, int column)`. Creates a coordinate by assigning the value of `row` to the row and `column` to the column.
- `int getRow() const`. *Getter* that returns the row.
- `int getColumn() const`. *Getter* that returns the column.
- `void setRow(int row)`. *Setter* that modifies the row.
- `void setColumn(int column)`. *Setter* that modifies the column.
- `bool isValid() const`. Returns `true` if the row and the column are greater than or equal to 0, and `false` otherwise.
- `void reset()`. Sets both attributes to -1, as if it were a coordinate just created with the no-argument constructor.
- `ostream& operator<<(ostream &os, const Coordinate &c)`. Output operator that displays the coordinate. For example, if the row is 2 and the column 7, it will display `[2,7]`. If the coordinate is invalid, it will display `[-,-]`. No newline should be added after showing this information on the screen.

3.2 Fighter

This class reflects the behavior of fighters, and it has several attributes (all of them private):

- `type`: a string (`string`) that indicates the type of fighter: `F-35B`, `EuroFighter`, etc.
- `speed`: the fighter's speed (must not be negative)
- `attack`: the fighter's attack capability (must not be negative)
- `shield`: the fighter's shield; if it reaches 0 or becomes negative, the fighter is considered destroyed
- `id`: an identifier that is unique for each fighter. When a new fighter is created, it is assigned the value of `nextId` as its identifier, and `nextId` is incremented; as shown in the diagram, initially `nextId` is 1
- `position`: a coordinate representing the fighter's position on the board; initially it will be an invalid coordinate (both row and column will be -1). It will be assigned a value when the fighter is placed on the board. When the fighter is destroyed and removed from the board, it must revert to an invalid coordinate
- `aircraftCarrier`: the name of the aircraft carrier to which the fighter belongs

In the UML diagram you can see the methods of this class. Many of them are simple getters that return the corresponding fields. For example, `getType` returns the `type` attribute of the object. The methods `addAttack`, `addSpeed`, and `addShield` act similarly to setters, but by adding the value received as an argument to the attribute (as the method name indicates). Moreover, if the speed (`speed`) or the attack capability (`attack`) become negative after the addition (the argument could be negative), they must be set to 0.

- `Fighter(string type, string aircraftCarrier)`. Constructor that initializes a fighter by assigning a value of 100 to speed, 80 to attack, and 80 to shield. In addition, it assigns the parameters to the attributes `type` and `aircraftCarrier`, and assigns the identifier (it increments `nextId` as well). The coordinate will be automatically initialized to the default value `(-1,-1)`, so nothing further is required. If `type` is an empty string, the `invalid_argument` exception must be thrown with the string `"Wrong type"`

- `void resetNextId()`. Static method that initializes the value of the static attribute `nextId` to 1. It should not be used in the assignment, but it is necessary for the autograder testing.

- `void resetPosition()`. Calls the `reset` method for the fighter's position.

- `int getDamage(int n) const`. Returns the damage inflicted by the fighter on the enemy fighter, which will always be `(n*attack)/300`. Since all numbers are integers, the division must be an integer division, and the result should be an integer.

- `int fight(Fighter *enemy)`. This method simulates a fight between two fighters, the one that receives the call (the *fighter*) and an enemy fighter (`enemy`). Before starting, if either of the two fighters has been destroyed, it returns 0; otherwise, it initiates a loop that will end when one of the two fighters is destroyed (a fight to the death). In each iteration of the loop, one of the fighters will attack the other. To do this, a random number `n` between 0 and 99 is obtained (by calling the function `getRandomNumber`), and this value is compared with a threshold u, which is calculated with the following formula:

  $$u = \frac{100v_1}{v_1 + v_2}$$

  where v_1 is the speed of the *fighter* and v_2 is the speed of the *enemy*. Since all values are integers, the division will be an integer (do not use floating-point numbers).

  If the threshold u is greater than or equal to the random number `n`,[1] then the attacker will be the *fighter* and the enemy's shield will be reduced by the damage caused by the fighter (which will be the value returned by the method `getDamage` passing `n` as a parameter). Otherwise, the attacker will be the *enemy*, and the fighter's shield will be reduced by the damage caused by the *enemy* (also calling the enemy's `getDamage`, but in this case passing `100-n` as a parameter).

  If a fighter is destroyed after an attack, the fight will end, and the method will return 1 if the *fighter* has won or -1 if the *enemy* has won. If neither fighter is destroyed, the loop will continue, and another random number will be obtained.

  For example, if $u = 50$ (both fighters have the same speed) and the first random number n is 80, 5 will be subtracted from the shield of the fighter (`this`); if the next number is 88, 3 will be subtracted again from the fighter, and if the next is 13, 3 will be subtracted from the enemy fighter's shield (`enemy`). The loop will continue until one of the fighters is destroyed.

  IMPORTANT: Make sure that you do not call `getRandomNumber` more than once within the loop. If you make additional calls, the result of this fight (and subsequent ones) will probably differ, and your assignment will not work correctly.

- `bool isDestroyed() const`. Returns `true` if the attribute `shield` is 0 or less, and `false` otherwise.

- `ostream& operator<<(ostream &os, const Fighter &f)`. Output operator that displays the fighter's data. For example, if an `F-35B` placed at position (2,3) has the identifier 27, speed 100, attack 80, and a shield value of 36 remaining, what should be displayed is:

  Terminal
  ```
  (F-35B 27 [2,3] {100,80,36})
  ```

  If the fighter does not have an assigned position, `[-,-]` will simply appear instead of `[2,3]` (the output operator of `Coordinate` will handle that). No newline should be added at the end.

[1] With this formula, the faster fighter will have a higher value of u and, therefore, a greater probability of attack.

3.3 AircraftCarrier

This class manages aircraft carriers, which will have several attributes (all of them private):

- `name`: the name of the aircraft carrier
- `wins`: victories obtained by the aircraft carrier's fighters
- `losses`: defeats of the aircraft carrier's fighters
- `fleet`: the fleet of fighters of the aircraft carrier (you must use a `vector<Fighter *>` to store the fighters)

In addition to the trivial getters, the methods of this class are:

- `AircraftCarrier(string name)`. Constructor that initializes the aircraft carrier's data (obviously, the attributes *wins* and *losses* must be initialized to 0). If the name is an empty string, the `invalid_argument` exception must be thrown with the string `"Wrong name"`
- `vector<Fighter *> getFleet() const`. Returns the fleet of fighters of the aircraft carrier; it will be used only in unit tests.
- `void addFighters(string fd)`. Given a string such as `"5/F-35B:12/F-18:3/A6:2/F-35B"`, this method should construct the indicated fighters and add them to the aircraft carrier's fleet. The number of fighters will appear at the beginning, separated from the type by "/". If there is more than one type of fighter, they will be separated by ":". In the example string, 5 `F-35B` fighters, 12 `F-18` fighters, 3 `A6` fighters, and another 2 `F-35B` fighters would be created. You can assume that the string has no errors, so it is not necessary to validate it. If the string is empty, the method will do nothing.
- `void updateResults(int r)`. This method should update the values of *wins* or *losses* depending on the value of the argument *r*; if it is 1, *wins* is incremented; if it is -1, *losses* is incremented. If *r* has any other value, the method does nothing.
- `Fighter *getFirstAvailableFighter(string type) const`. Returns the first fighter (not destroyed and not positioned on the board) in the fleet of the type indicated by the argument *type*. If *type* is an empty string, it returns the first fighter (not destroyed and not positioned) of any type. If there are no non-destroyed fighters or if there are no fighters at all, it returns `NULL` (or `nullptr`).
- `int purgeFleet()`. Removes the destroyed fighters from the fleet, and returns the number of fighters removed.
- `void showFleet() const`. Displays the complete fleet, showing one fighter per line. If a fighter has been destroyed, the string `"(X)"` will be appended at the end of the line to indicate that it was destroyed, as in this example:

Terminal

```
(F-35B 24 [-,-] {100,80,-24}) (X)
(F-35B 25 [1,3] {100,80,80})
(F-35B 26 [-,-] {100,80,80})
(F-35B 27 [-,-] {100,80,80})
(F-35B 28 [-,-] {100,80,80})
```

  All fighters in the fleet must be displayed, including those that are destroyed and those that are positioned. If the fleet has no fighters, nothing is shown.

- `string myFleet() const`. Returns a string with the format accepted by the method `addFighters` containing the fighters (destroyed or not) in the fleet. For example, if an aircraft carrier has 7 `F-35B` fighters and 1 `Super Hornet`, it would return the string `"7/F-35B:1/Super Hornet"` (or the string `"1/Super Hornet:7/F-35B"`, depending on whether the first fighter is an `F-35B` or a `Super`

`Hornet`). If the fleet has no fighters, it returns an empty string. Note that the fighters must be shown in the order they appear in `fleet` and that all fighters of the same type must be accumulated; for example, if an aircraft carrier has in its fleet 3 `F-35B`, 2 `F-18`, 4 `F-35B`, 5 `Super Hornet`, and 2 `F-35B`, the returned string must be `"9/F-35B:2/F-18:5/Super Hornet"` because the first type of fighter that appears in the fleet is the `F-35B`, then the `F-18`, and finally the `Super Hornet`.

- `ostream& operator<<(ostream &os, const AircraftCarrier &a)`. Output operator that displays the data of the aircraft carrier. For example, if an aircraft carrier named USS Abraham Lincoln has achieved 35 victories and suffered 10 defeats and has a fleet of 12 `F-35B` fighters and 7 `F-18` fighters, the following text would be displayed without adding a newline at the end:

Terminal

```
Aircraft Carrier [USS Abraham Lincoln 35/10] 12/F-35B:7/F-18
```

3.4 Board

This class represents the square board on which the game will be played, and it has two attributes:

- `size`: the size of the board (must always be positive). The coordinates within the board will range from 0 to *size-1*.
- `board`: a square matrix of pointers to fighters to store the fighters in positions on the board. It should be declared as:

```
vector<vector<Fighter *>> board;
```

The methods of this class are:

- `Board(int size)`. Constructor that initializes the board's data. If the value of `size` is not greater than `0`, the `invalid_argument` exception must be thrown with the string `"Wrong size"`; if it is valid, a square matrix of size `size` must be created, and all pointers should be initialized to `NULL` or `nullptr` (fighters will be stored later).
- `Fighter *getFighter(Coordinate c) const`. Getter that returns the content of the board at the indicated position, which will be the fighter occupying it, or `NULL` (`nullptr`) if there is nothing.
- `bool inside(Coordinate c) const`. Returns `true` if the coordinate is inside the board, and `false` otherwise. Coordinates are inside the board if the components of the coordinate are between 0 and *size-1* (both values included).
- `int launch(Coordinate c, Fighter *f)`. Attempts to place a fighter in a position on the board (if it is not already positioned). Obviously, if the coordinate is not inside the board, it does nothing, as it also does nothing if the position is occupied by another fighter from the same aircraft carrier. However, if the position is occupied by a fighter from another aircraft carrier (an enemy fighter), the new fighter will attack and fight with the one that occupied that position. At the end of the fight (in which one of the fighters always wins and the other is destroyed), the winning fighter will remain in that position, and the losing fighter will have its position reset. Finally, if the position was empty, the fighter will be placed in that position (also updating the fighter's position with the corresponding setter).

  For example, if there is an `SF-1` at position `[2,3]` on the board and we want to launch an attacking fighter `F-35B` to that position, the `F-35B` will fight with the `SF-1`, and two situations can happen:

  - If the `F-35B` wins, it will be assigned that position on the board (and its position will be updated), and the `SF-1` will have its position reset.
  - If the `SF-1` wins, neither fighter will change position (the `F-35B` should not have a position to be launched, so it remains unchanged).

The method must return the result of the fight with the enemy fighter: a 1 if the attacking fighter wins or a -1 if the enemy (the one on the board) wins. In any other case (if it was already on the board, or there is no fighter, or it is a friendly fighter, or the coordinate is not on the board), the method will return 0.

Finally, if the method returns the result of the fight, that value must be used elsewhere in the code (e.g., in `prac3.cc`) to update the win/loss statistics of the aircraft carriers for both fighters.

4 Main Program

The main program is in the file `prac3.cc` published on the course's Moodle. It is simply an example of how the classes you have created can be used in a program. You can modify it as you wish to test your classes; it should not be submitted along with the other files.

This file contains code to create several aircraft carriers and fighters, and simulates a battle on the board:

```
#include <iostream>
#include "AircraftCarrier.h"
#include "Board.h"

using namespace std;

int main(){
     srand(888);

     Board board(5);  // 5x5

     AircraftCarrier one("USS Acme");
     one.addFighters("3/F-35B:2/F-18:3/A6");

     AircraftCarrier two("Spectra One");
     two.addFighters("4/SF-1:3/SB-3:2/SI-6B");

     // position some fighters on the board
     Coordinate c1(2,3);
     Fighter *f1 = one.getFirstAvailableFighter("F-18");
     board.launch(c1,f1);

     Coordinate c2(3,2);
     Fighter *f2 = two.getFirstAvailableFighter("");
     board.launch(c2,f2);

     cout << "After launching one fighter of each aircraft:" << endl;
     cout << one << endl;
     one.showFleet();
     cout << two << endl;
     two.showFleet();

     Fighter *f3 = one.getFirstAvailableFighter("");
     int result = board.launch(c2,f3);
     if (result != 0)
     {
       one.updateResults(result);
       two.updateResults(-result);
     }
     cout << "After a fight between two fighters:" << endl;     cout << one << endl;
     one.showFleet();
     cout << two << endl;
     two.showFleet();
```

```
    return 0;
}
```

The output would be:

```
After launching one fighter of each aircraft:
Aircraft Carrier [USS Acme 0/0] 3/F-35B:2/F-18:3/A6
(F-35B 1 [-,-] {100,80,80})
(F-35B 2 [-,-] {100,80,80})
(F-35B 3 [-,-] {100,80,80})
(F-18 4 [2,3] {100,80,80})
(F-18 5 [-,-] {100,80,80})
(A6 6 [-,-] {100,80,80})
(A6 7 [-,-] {100,80,80})
(A6 8 [-,-] {100,80,80})

Aircraft Carrier [Spectra One 0/0] 4/SF-1:3/SB-3:2/SI-6B
(SF-1 9 [3,2] {100,80,80})
(SF-1 10 [-,-] {100,80,80})
(SF-1 11 [-,-] {100,80,80})
(SF-1 12 [-,-] {100,80,80})
(SB-3 13 [-,-] {100,80,80})
(SB-3 14 [-,-] {100,80,80})
(SB-3 15 [-,-] {100,80,80})
(SI-6B 16 [-,-] {100,80,80})
(SI-6B 17 [-,-] {100,80,80})

After a fight between two fighters:
Aircraft Carrier [USS Acme 1/0] 3/F-35B:2/F-18:3/A6
(F-35B 1 [3,2] {100,80,4})
(F-35B 2 [-,-] {100,80,80})
(F-35B 3 [-,-] {100,80,80})
(F-18 4 [2,3] {100,80,80})
(F-18 5 [-,-] {100,80,80})
(A6 6 [-,-] {100,80,80})
(A6 7 [-,-] {100,80,80})
(A6 8 [-,-] {100,80,80})

Aircraft Carrier [Spectra One 0/1] 4/SF-1:3/SB-3:2/SI-6B
(SF-1 9 [-,-] {100,80,-3}) (X)
(SF-1 10 [-,-] {100,80,80})
(SF-1 11 [-,-] {100,80,80})
(SF-1 12 [-,-] {100,80,80})
(SB-3 13 [-,-] {100,80,80})
(SB-3 14 [-,-] {100,80,80})
(SB-3 15 [-,-] {100,80,80})
(SI-6B 16 [-,-] {100,80,80})
(SI-6B 17 [-,-] {100,80,80})
```