# BOOM - Battleships with Object-Oriented Makeover

By Richard Pawson (rpawson@stowe.co..uk)

This project takes the original Battleships program as used as a skeleton program for AQA, and gives it a comprehensive object-oriented makeover, as well as other code-level improvements.

(The initiative for this project came from someone else, who wrote, and shared, a partially object-oriented version of Battleships. I regret that I can’t now find the source of that code, so am unable to credit it properly. My approach takes the object-oriented design principles much further - but I should stress that this is *not* intended as any criticism of that piece of work.)

BOOM is NOT intended for teaching in conjunction with the original skeleton program, but rather as a separate exercise in object-oriented design and coding.

Also, note that I have not stayed within the same constraints as the original:

* BOOM is written in C#, and while it *could* be translated to other languages, I have not explicitly attempted to write it in such a way as to facilitate this.
* BOOM follows the original game fairly closely, though with a couple of small improvements only: a Bomb/Missile option, and notification when a ship is sunk.

Finally, before reading further, I recommend that you

* Familiarise yourself with the original AQA Battledships game and code
* Read my resource (on CAS): [A detailed example of refactoring existing code in C#](http://community.computingatschool.org.uk/resources/4824), as I have re-use a lot of the same patterns as in that project, and even re-used code. Reading this is particularly important if you are not familiar with the concepts of ‘separation of concerns’ and ‘dependency injection’. In that paper I explained all of the changes made; for BOOM I have not done this.

## Major changes to the object model

* There are now two types of weapon – Missile and Bomb - that both implement an interface (IWeapon) in order to achieve polymorphism. (This could also have been done by making them inherit from an abstract superclass Weapon, but my preference is always to use interfaces unless there is an actual need to inherit implementation).
* A great deal of responsibility has been shifted from the GameBoard object to the Ship objects. The latter are responsible for knowing where they are, and therefore whether they occupy a given location, and also to keep track of how many times (and where) they have been hit, which in turn allows them to know when they have been sunk. The GameBoard is now responsible for interfacing with the ships, and for keeping track where shots have been made and their outcome, in terms of Hit and Miss. There is a very small level of data duplication left, in that a hit is now recorded both on the Board and the Ship; removing this (by referring indirectly to the Ships each time the board is re-drawn) would improve the purity of the model, but is left as an exercise for the reader (unless I get fed up with it!)
* The original game loaded a ‘training game’ from a text file. This has been replaced with the concept of ‘data fixtures’ in code, which facilitates testing (see below). Note, too, these data fixtures are now just arrays of Ships, not of board locations
* The object model makes use of two Enums (Orientations, and SquareValues). This makes code easier to read, and also much safer - since you cannot then accidentally use an illegal values (such as a mis-typed character). So the board is now just a two-dimension array of SquareValues, not of chars.
* The board size is now variable.

## Major changes to the project structure

In line with the approach that I took to refactoring ‘PredatorPrey’ (the link included above), and, indeed, for most applications that I write, I have split the solution into multiple projects in order to achieve separation of concerns, and improve maintainability, flexibility, and testability. The following is an outline explanation only, explaining the different projects within solution, in a sensible order:

* **Boom.Model** contains the ‘domain model’. This includes the classes: Ship, GameBoard, Missile, Bomb, the interface IWeapon, plus the enums mentioned above. One of the aims was to keep all user interface code out of this project, and put it in …
* **Boom.ConsoleUI**. This is the Console user interface, following, fairly closely, the original UI from the Battleships program. The intent here was to try to keep any domain logic *out of* this project. Most of the calls to the Console were fairly easy to extract, however, some remained deep within the domain code (e.g. the messages giving the outcome from firing a weapon). This could have been factored out by going for an ‘event driven’ approach, but I didn’t want to do that. Instead, to make the Model fully independent of the user interface I re-used the pattern of an ‘injected’ service, defined by an interface, within …
* **TechnicalServices*.*** This is best understood by reading my paper on refactoring PredatorPrey. Not only have I used the same patterns here in BOOM, but I have deliberately re-used the same actual code - indeed that is the reason why this project’s name does *not* start with ‘Boom’. In that paper, I flagged the idea that it would be possible to create new implementations of the two ‘service’ interfaces and indeed that is exactly what I have done here: a new implementation of ILogger (ReadbleLogger) and a new implementation of IRandomGenerator (PredictableRandomGenerator) both for use, *inter alia*, in …
* **Boom.Test**. This project uses the Microsoft Unit Test framework to implement automated tests. The tests provide reasonably good coverage of the domain code functionality, though by no means 100%. This proved invaluable to me as I was developing the domain code, and would be even more useful to anyone modifying or extending the application in future, to check, within seconds, whether they had accidentally broken existing functionality. Note that though these tests use the Unit Test framework, they are not *strictly* ‘unit tests’ - they are really ‘functional tests’. Unit tests, strictly speaking, test a single function or method; my test multiple methods, indeed multiple classes, working together. But this allows me to test complete scenarios, even playing a complete game in one test to test the detection of all ships being sunk. Within the professional development community, there is much debate about the value of unit tests; I am on the side that favours functional tests - and write strict unit tests only for certain kinds of functions. Note that this commitment to reasonably comprehensive automated tests is only possible because of separating the model from the user interface. The latter, while not impossible to test, is very much harder to test in an automated fashion. Both the Boom.Test project, and the Boom.ConsoleUI projects rely on …
* **Boom.DataFixture*.*** This contains example data used both during real play, and in automated testing. One of the data fixtures is the replacement for the ‘Training Game’, which was loaded from a text file in the original Battleships game. By keeping the fixture in code, it performs much faster, which is important for running the tests, each of which must create the game and load any fixture from scratch. Boom.ConsoleUI also has a menu option for loading the training game this way, in line with the original. Finally the principle ‘separation of concerns’ represented by the use of the various projects listed here, pays another huge dividend: it allows us to create an entirely new user interface re-using all of the above, and with minimal duplication of functionality. Many such user interfaces are possible, e.g. a web user interface (written, for example, using the ASP.NET MVC framework). I have added just one more …
* **Boom.WinFormsUI**. This is a simple graphical user interfaces, implemented using Windows Forms. It could be significantly improved, but serves to demonstrate the basic idea. Like the ConsoleUI, the intent is to keep domain logic out of the UI and delegate as much as possible to the Model and other projects. Writing Win Forms code is not easy (at least, I don’t find it so), but the nice thing here is just how little code is used to make it work.

## Using the BOOM application in teaching

My intent in writing BOOM was to provide a non-trivial example of a fully object-oriented application that pupils could explore, modify and extend, and that would clearly demonstrate the benefits of the OO approach. I have started to use this with my Lower 6th pupils, on the AQA syllabus. As well as reinforcing the principles of OOP, my secondary intent is to help teach good (C#) programming practices, many of which the pupils can (and, IMO, *should*) apply in their NEA projects.

When I have shown this to pupils, I have not revealed the whole solution at once - starting, for example, *without* the Boom.Test or the Boom.WinFormsUI projects included, and either got the pupils to start developing both, or demonstrated their development (with students following along adding code on their own machines).

In terms of OOP, it is also useful to return to the original idea of AQA skeleton programs and pose specific challenges to pupils to modify/extend the model. Examples include:

* Adding a Torpedo as a third type of weapon.
* Specifying a limit for how many times each type of weapon can be used.
* Adding a Search Light option that, *very briefly*, ‘illuminates’ an area of the board, which the player must remember and shoot at.
* Making ships mobile - until they are hit! With each player move, any ships not yet hit move one square in a random direction, subject to checking that the new position is still legal. This means the player may need to target the same squares multiple times.