

## Design Pattern: Builder Pattern

The goal of the builder design pattern is to separate the construction of an object from its representation. This means we can create a builder that adds new features without directly modifying the machine classes, hence avoiding the explosion of classes.

- *LowPowerMachine*, *MediumPowerMachine* and *HighPowerMachine* are what we call the base classes. These classes inherit the description and cost logic from the *AmMachine* parent class.
- *AmMachineFeatures* acts as a container for optional features (e.g. *HasQuadLaser*). It includes methods to calculate the additional cost of the feature and modify the description of the machine. Because features are in their own class, you can easily add or remove them without modifying existing machine classes.
- *CustomAmMachine* inherits methods from the *AmMachine* parent class but wraps a base machine together with a set of features from *AmMachineFeatures*. It overrides the description and cost methods to combine the base machine's values with the selected features.
- The *AmMachineBuilder* starts with a base machine from *AmMachine* and toggles the feature flags in *AmMachineFeatures* (e.g. *AddQuadLaser()* sets *HasQuadLaser* to true). Finally, it builds a *CustomAmMachine* that merges the base machine with those features.

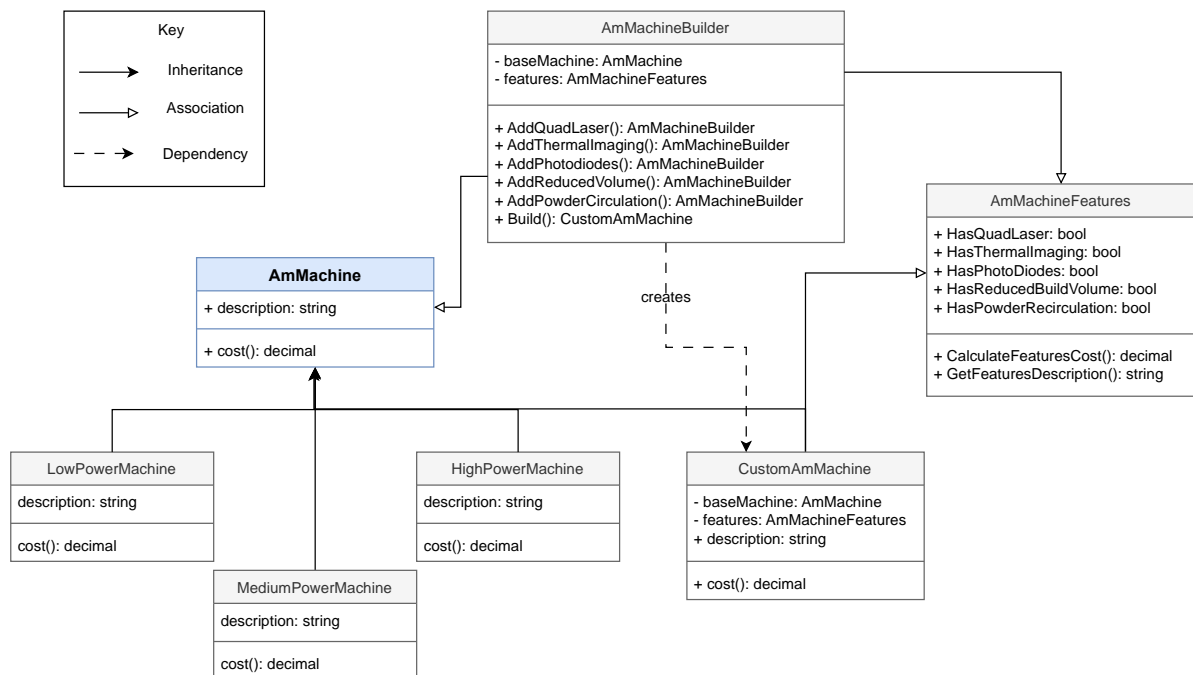
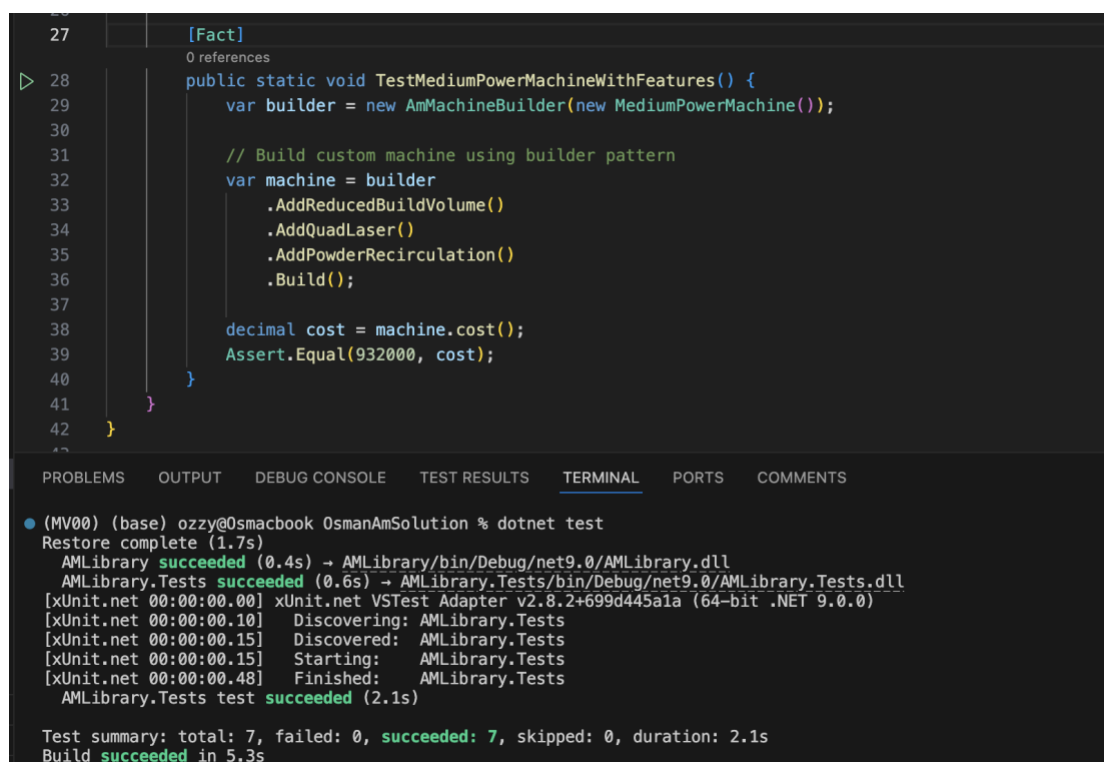


Figure 1 – Class Diagram

When applying the builder pattern, we also apply several design principles. We apply the **Open-Closed** principle which asserts that classes should be open for extension and closed for modification. Using this approach, if you need to add new features in the future, you only update *AmMachineFeatures* (for cost and description logic) and possibly add a new method in *AmMachineBuilder*. Another design principle we use is the **Single Responsibility** Principle, which ensures that each class has one responsibility or one reason to change. In our example, the base machine classes are responsible for their own cost and description. The *AmMachineFeatures* is responsible for holding the features and calculating their cost. The *AmMachineBuilder* is responsible for assembling the custom machine using the accumulated features. The *CustomAmMachine* is responsible for combining the base machine's cost and description with the cost and description of the new features. Therefore, each class has a single responsibility.

## Unit Test

To examine the functionality of our class code, we used a unit test to assert that the cost of a medium power machine with reduced build volume, a quad laser and a powder recirculation system is £932,000:



```
27 [Fact]
28 0 references
29 public static void TestMediumPowerMachineWithFeatures() {
30     var builder = new AmMachineBuilder(new MediumPowerMachine());
31
32     // Build custom machine using builder pattern
33     var machine = builder
34         .AddReducedBuildVolume()
35         .AddQuadLaser()
36         .AddPowderRecirculation()
37         .Build();
38     decimal cost = machine.cost();
39     Assert.Equal(932000, cost);
40 }
41 }
42 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TEST RESULTS TERMINAL PORTS COMMENTS

```
● (MV00) (base) ozzy@Osmacbook OsmanAmSolution % dotnet test
Restore complete (1.7s)
  AMLibrary succeeded (0.4s) → AMLibrary/bin/Debug/net9.0/AMLibrary.dll
  AMLibrary.Tests succeeded (0.6s) → AMLibrary.Tests/bin/Debug/net9.0/AMLibrary.Tests.dll
[xUnit.net 00:00:00.00] xUnit.net VSTest Adapter v2.8.2+699d445a1a (64-bit .NET 9.0.0)
[xUnit.net 00:00:00.10] Discovered: AMLibrary.Tests
[xUnit.net 00:00:00.15] Discovered: AMLibrary.Tests
[xUnit.net 00:00:00.15] Starting: AMLibrary.Tests
[xUnit.net 00:00:00.48] Finished: AMLibrary.Tests
  AMLibrary.Tests test succeeded (2.1s)

Test summary: total: 7, failed: 0, succeeded: 7, skipped: 0, duration: 2.1s
Build succeeded in 5.3s
```

Figure 2 – Unit Test

## Alloy Optimisation

For the Alloy Optimisation problem, our objective is to maximise the creep resistance of an alloy while ensuring the total cost does not exceed a given budget. Initially, a grid search approach was considered, where we evaluate all possible combinations of element percentages within their specified ranges. While this approach guarantees an optimal solution, the search space grows exponentially with the number of elements:  $O(k^n)$ , where  $k$  is the number of steps and  $n$  is the number of elements. Hence, we opted for another approach.

### Approach:

To overcome the inefficiencies of grid search, we implemented a linear programming (LP) approach using the Google OR-Tools library. LP uses linear equations as constraints to maximise an objective function. In our case, the objective function is the creep resistance (CR) equation:

$$\text{Maximise } CR = \sum_{i \in E} \alpha_i x_i$$

Where  $\alpha_i$  is the creep coefficient and  $x_i$  is the atomic percentage of element  $i$ . The base element does not contribute to the creep resistance so it was excluded from the objective function but it was included in the constraints since it has a cost. The 1<sup>st</sup> constraint is that the total cost must not exceed our budget,  $maxCost$ :

$$\sum_{i \in A} \frac{c_i x_i}{100} \leq maxCost$$

Where  $c_i$  is the element cost per kg. The 2<sup>nd</sup> constraint is that total percentage of elements must equal 100%:

$$\sum_{i \in A} x_i = 100$$

We also need to have discrete step sizes for element percentages:

$$x_i = min_i + k_i \times step_i \text{ where } k_i \in \{0, 1, \dots, \lceil \frac{max_i - min_i}{step_i} \rceil\} \text{ and } min_i \leq x_i \leq max_i$$

Since element percentages can only increase in discrete steps (e.g., 0.5%, 1%), we used integer variable  $k_i$  to represent the number of steps for each element. This makes the problem a Mixed Integer Linear Programming (MILP) problem. In MILP, the solver is called SCIP (Solving Constraint Integer Programs) and it uses an algorithm called Branch and Bound. This involves dividing the problem into smaller subproblems by fixing integer variables to specific values (branching) and solving the LP for each subproblem to get an upper and lower bound for the objective function (bounding). This iterative process continues, narrowing down the search space until an optimal solution is found or all branches are explored or discarded. The main advantage of using this solver is that it takes into account the step sizes for each element percentage and efficiently traverses the solution space to find an optimal composition within the defined constraints.

## Implementation:

To implement the MILP solution in code, three main classes were created. The *Element* class acts like a data container storing properties like alpha, cost and range constraints:

```
// Element class stores element properties
19 references
public class Element {
    15 references
    public string Name {get; set;}
    12 references
    public decimal Alpha {get; set;}
    13 references
    public decimal Cost {get; set;}
    15 references
    public decimal MinPercentage {get; set;}
    12 references
    public decimal MaxPercentage {get; set;}
    15 references
    public decimal StepSize {get; set;}
}
```

Figure 3 – Element Class

The *AlloyOptimiser* class takes these elements plus the *maxCost*, then runs the MILP optimisation process to find an optimal distribution of element percentages within the specified constraints. The class also takes in the base element as an argument. In earlier implementations, we assumed the base element was always the one with an alpha value of zero (e.g., Nickel). However, elements with a zero or near-zero alpha value might not necessarily be the base element and may still contribute to the alloy. To address this, we changed the base element to be an argument for the *AlloyOptimiser* class and the solver automatically maximises the creep resistance using all elements except the designated base element.

In terms of structure, the optimiser first calculates how many discrete steps are possible using *MinPercentage*, *MaxPercentage* and *StepSize*. An integer variable *varSteps* is then created, constrained to range from zero up to the number of possible steps. The objective function is then maximised via *objective.SetMaximisation()* before applying the percentage constraint and the cost constraint in that order.

Finally, the *Alloy* class stores the composition of the solution, mapping each element to a final percentage. It includes methods to calculate creep resistance and cost that can be called after the optimiser returns its results. This is more of a validation step to ensure the final composition is valid. Essentially, the *Alloy* class encapsulates the final distribution of elements, allowing you to examine the key alloy properties such as total cost or creep resistance. Initially this was part of the optimiser class but was then separated into its own class to adhere with the single responsibility principle.