SSE 661

Software Architecture

Project #1

by

Jason Payne

June 2, 2015

TABLE OF CONTENTS

Topics Covered	Topic Examples
Software Architecture	• Class Text Overview • Pipes & Filters Pattern

1. Software Architecture Text Overview

The following table is a high-level overview of the text that will be used for SSE 661: Software Architecture.

Volume 1	**Chapter 2: Architectural Patterns** (skeleton of overall system architecture)	From Mud to Structure	- Layers - Pipes & Filters - Blackboard
		Distributed Systems	- Broker
		Interactive Systems	- Model-View-Controller - Presentation-Abstraction-Control
		Adaptable Systems	- Microkernel - Reflection (Open Implementation, Meta-Level Architecture)
	Chapter 3: Design Patterns (addresses problems encountered AFTER the system architecture has been specified)	Structural Decomposition	- Whole-Part - Composite
		Organization of Work	- Master-Slave - Chain of Responsibility - Command - Mediator
		Access Control	- Proxy - Façade - Iterator
		Management	- Command Processor - View Handler - Memento
		Communication	- Forwarder-Receiver - Client-Dispatcher-Server - Publisher-Subscriber (GoF: Observer)

Volume 2	Concurrent & Networked Objects	Service Access & Configuration Patterns	- Wrapper Façade - Component Configurator (Service Configurator) - Interceptor - Extension Interface
		Event Handling Patterns	- Reactor (Dispatcher, Notifier) - Proactor - Asynchronous Completion Token (Active Demultiplexing, Magic Cookie) - Acceptor-Connector
		Synchronization Patterns	- Scoped Locking (Synchronized Block, Resource-Acquisition-is-Initialization, 1 Guard, Execute Around Object) - Strategized Locking - Thread-Safe Interface - Double-Checked Locking Optimization (Lock Hint)
		Concurrency Patterns	- Active Object (Concurrent Object) - Monitor Object (Thread-safe Passive Object) - Half-Sync / Half-Async (Thread-Local Storage) - Leader / Followers - Thread-Specific Storage

2. Pipes & Filters Pattern

The Pipes & Filters Design Pattern is an architectural design patter aimed at reducing duplicated code contained within rigid and hard to read classes. These rigid classes are typically large with monolithic, repeated behavior and logic which is a direct contradiction of the Single Responsibility and Don't Repeat Yourself principles. The Pipes & Filters pattern addresses this problem by decomposing larger tasks into a series of unique objects, or filters. In doing this, the performance, scalability, and reusability of the application are improved by allowing the filter objects that perform the processing to be deployed and scaled independently. For this section, an example of a rigid, monolithic application is presented as an exercise for the application of the Pipes & Filters pattern.

2.1 Applied Example

In this example, the logical representation of an automobile manufacturing process is presented as a console application. The intended goal of the application is to be able to produce different types of automobiles using different automobile factories with unique and independent build processes. However, as is quickly seen, the original solution meets the previously mentioned requirements, but fails to implement a flexible design. Because of this, the factories must be re-designed in order to accommodate changing safety protocols and future growth opportunities within the company. The original solution is presented first followed by the redesigned solution utilizing the *Pipes & Filters* pattern.

2.1.1 Original Solution

The design and implementation of the original solution takes on a very basic design concept where different factories are created to produce different types of automobiles (in this case, a car and a truck). So, to produce a car, a `CarFactory` object must be created and then have the `CarFactory` produce the car by calling its `Create` method (and likewise to produce a truck). It is at this point that a variety of tasks of varying complexity are performed to produce a specific `Automobile` object. As previously mentioned, a straightforward, but inflexible approach to implementing this application uses large, monolithic "modules" as the process for creating the `Automobile` objects. The consequences of this approach are likely to reduce the opportunities for refactoring the code, optimizing it, or reusing it if parts of the same processing can be utilized by other automobile factories. *Figure 2-1* illustrates the class diagram for this initial version of the automobile factory solution.

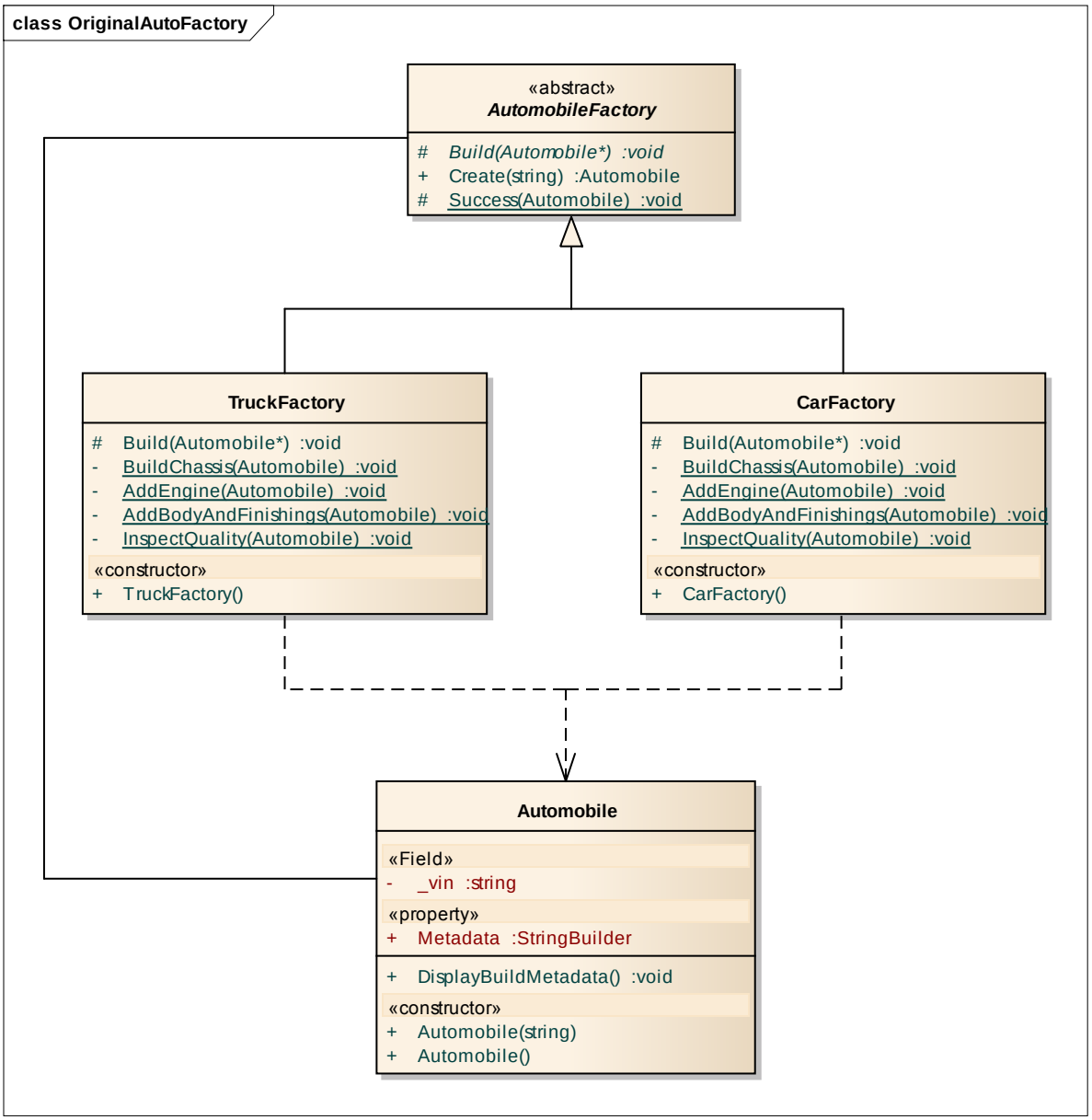

Figure 2-1: *Class Diagram of Original Solution*

As can be seen from the class diagram, each instance of an `AutomobileFactory` inherits a `Create` method that is used by the calling code to initiate the process of producing the specified `Automobile` object. While this `Create` method is inherited by the child classes, it is not overridable because the `AutomobileFactory` class has been designed according to the Template Method design pattern which allows the `AutomobileFactory` class to define the procedural steps used in building the `Automobile` objects while allowing different implementations – such as `CarFactory` and `TruckFactory` – to exist in order to build different types of `Automobile` objects. The different procedures involved in building the different `Automobile` objects are defined by the specific implementation of the inherited `Build` method.

Some of the tasks that each factory performs are functionally very similar, but cannot share the similar code due to the current design. The code that implements these tasks is closely coupled within each module, and this code has been developed with little or no thought given to reuse or scalability. This becomes a more prominent issue if/when the business requirements change, such as implementing new safety protocols in the build process and/or creating new factories for new types of `Automobile` objects.

Automobile

This class is the logical representation of an automobile. It is the end product produced by each `AutomobileFactory`. For the sake of this example, it provides a simple `StringBuilder` property for illustrating the different stages of the build process. A more robust version would include logical representations of a real-world automobile such as tires, engines, transmissions, etc.

```
public class Automobile
{
    private string _vin;

    public StringBuilder Metadata { get; private set; }

    public Automobile(string vin)
    {
        _vin = vin;
        Metadata = new StringBuilder();
    }

    public void DisplayBuildMetadata()
    {
        Console.WriteLine(_vin);
        Console.WriteLine("===========");
        Console.WriteLine(Metadata.ToString());
    }
}
```

AutomobileFactory

This class is the logical representation of an automobile factory. It is designed according to the Template Method pattern so that the proper order of logical steps required to start and invoke the build process can be defined at one place in the solution. The abstract `Build` method is provided to child classes so that different types of factories can be created for the purposes of creating different types of `Automobile` objects.

```
public abstract class AutomobileFactory
{
    public Automobile Create(string vin)
    {
        var automobile = new Automobile(vin);
        Build(ref automobile);
        return automobile;
    }

    protected abstract void Build(ref Automobile a);
```

```
    protected static void Success(Automobile a)
    {
        a.Metadata.AppendLine("SUCCESS!");
    }
}
```

CarFactory

This class is the logical representation of a factory of cars. It inherits from the abstract `AutomobileFactory` class and must therefore implement the `Build` method to define its specific process for creating a car. All of the implementation is self-contained and has the potential to be a large, monolithic class. For the sake of this example, the specific implementation details involved at each stage of the manufacturing process are symbolized by the ellipses.

```
public class CarFactory : AutomobileFactory
{
    protected override void Build(ref Automobile c)
    {
        BuildChassis(c);

        AddEngine(c);

        AddBodyAndFinishings(c);

        InspectQuality(c);
    }

    private static void BuildChassis(Automobile c)
    {
        c.Metadata.Append("Molding & welding chassis...");
        //...

        c.Metadata.Append("Adding front suspension...");
        //...

        c.Metadata.Append("Adding rear suspension...");
        //...

        c.Metadata.Append("Adding gas tank...");
        //...

        c.Metadata.Append("Adding rear axles...");
        //...

        c.Metadata.Append("Adding drive shafts...");
        //...

        c.Metadata.Append("Adding gear boxes...");
        //...

        c.Metadata.Append("Adding steering box components...");
        //...

        c.Metadata.Append("Adding wheel drums...");
        //...

        c.Metadata.Append("Adding braking system...");
        //...

        Success(c);
```

```
    }

    private static void AddBodyAndFinishings(Automobile c)
    {
        c.Metadata.Append("Adding body to frame...");
        //...

        c.Metadata.Append("Adding battery & spark plugs...");
        //...

        c.Metadata.Append("Adding tires...");
        //...

        c.Metadata.Append("Topping-off fluids...");
        //...

        Success(c);
    }

    private static void AddEngine(Automobile c)
    {
        c.Metadata.Append("Mating engine with transmission...");
        //...

        c.Metadata.Append("Attaching radiator...");
        //...

        Success(c);
    }

    private static void InspectQuality(Automobile c)
    {
        c.Metadata.Append("Inspecting quality...");
        //...
        Success(c);
    }
}
```

TruckFactory

This class is the logical representation of a factory of trucks. It inherits from the abstract `AutomobileFactory` class and must therefore implement the `Build` method to define its specific process for creating a truck. Note the different implementation details (highlighted) that are specific to the build process of a truck. Other than that, the build process logic is very similar between a `CarFactory` and a `TruckFactory`. All of the implementation is self-contained and has the potential to be a large, monolithic class. For the sake of this example, the specific implementation details involved at each stage of the manufacturing process are symbolized by the ellipses.

```
public class TruckFactory : AutomobileFactory
{
    protected override void Build(ref Automobile t)
    {
        BuildChassis(t);

        AddEngine(t);

        AddBodyAndFinishings(t);
```

```
        InspectQuality(t);
    }

    private static void BuildChassis(Automobile t)
    {
        t.Metadata.Append("Molding & welding chassis...");
        //...

        t.Metadata.Append("Adding trailer hitch...");
        //...

        t.Metadata.Append("Adding front suspension...");
        //...

        t.Metadata.Append("Adding rear suspension...");
        //...

        t.Metadata.Append("Adding gas tank...");
        //...

        t.Metadata.Append("Adding rear axles...");
        //...

        t.Metadata.Append("Adding drive shafts...");
        //...

        t.Metadata.Append("Adding gear boxes...");
        //...

        t.Metadata.Append("Adding steering box components...");
        //...

        t.Metadata.Append("Adding wheel drums...");
        //...

        t.Metadata.Append("Adding braking system...");
        //...

        Success(t);
    }

    private static void AddEngine(Automobile t)
    {
        t.Metadata.Append("Mating engine with transmission...");
        //...

        t.Metadata.Append("Attaching radiator...");
        //...

        Success(t);
    }

    private static void AddBodyAndFinishings(Automobile t)
    {
        t.Metadata.Append("Adding body to frame...");
        //...

        t.Metadata.Append("Adding bed to frame...");
        //...
```

```
        t.Metadata.Append("Adding battery & spark plugs...");
        //...

        t.Metadata.Append("Adding tires...");
        //...

        t.Metadata.Append("Topping-off fluids...");
        //...

        Success(t);
    }

    private static void InspectQuality(Automobile t)
    {
        t.Metadata.Append("Inspecting quality...");
        //...

        Success(t);
    }
}
```

Program (Test Driver Class)

This class is the main driver of the console application. It is the calling object that creates the different types of `AutomobileFactory` objects for the sake of creating specific types of `Automobile` objects like a car and a truck. For this example, the program simply creates a `CarFactory`/`TruckFactory` object for creating a car/truck and displays the results of the build process to the console.

```
public class Program
{
    private static void Main(string[] args)
    {
        Log.Initialize();

        // Create a new car
        var carFactory = new CarFactory();
        var car = carFactory.Create("C-123ABC");
        car.DisplayBuildMetadata();

        // Create a new truck
        var truckFactory = new TruckFactory();
        var truck = truckFactory.Create("T-789XYZ");
        truck.DisplayBuildMetadata();

        Console.ReadKey();
        Log.Close();
    }
}
```

Output:

```
C-123ABC
===========
Molding & welding chassis...Adding front suspension...Adding rear suspension...Adding
gas tank...Adding rear axles...Adding drive shafts...Adding gear boxes...Adding
steering box components...Adding wheel drums...Adding braking system...SUCCESS!
Mating engine with transmission...Attaching radiator...SUCCESS!
```

```
Adding body to frame...Adding battery & spark plugs...Adding tires...Topping-off
fluids...SUCCESS!
Inspecting quality...SUCCESS!

T-789XYZ
===========
Molding & welding chassis...Adding trailer hitch...Adding front suspension...Adding
rear suspension...Adding gas tank...Adding rear axles...Adding drive shafts...Adding
gear boxes...Adding steering box components...Adding wheel drums...Adding braking
system...SUCCESS!
Mating engine with transmission...Attaching radiator...SUCCESS!
Adding body to frame...Adding bed to frame...Adding battery & spark plugs...Adding
tires...Topping-off fluids...SUCCESS!
Inspecting quality...SUCCESS!
```

2.1.2 Refactored Solution

To properly refactor the original solution, a set of design goals were established in order to accommodate new requirements. Those goals were:

1. Decompose the monolithic `Build` methods of the factory classes into a set of discrete components (or filters), each of which performs a single task.
2. Reduce duplicated code.
3. Increase design flexibility in order to remove, replace, or integrate additional build steps if the processing requirements change.
4. Increase scalability so that certain build steps can run in parallel processes (such as building the engine while the chassis is being created).

The Pipes & Filters pattern helped meet all of these goals by decomposing the build steps into discrete components (filters) that perform a single task. Each filter executes a specific step in the build process yielding a more complete version of the `Automobile` object than when it first entered the filter. These filters can then be combined together to form a logical assembly line that makes it trivial to remove, replace, or integrate additional components if the processing requirements change. This also allows for parallel processing of logical assembly lines that could take longer to process. A good context-specific example is for the case where the engine must be built at the same time that the chassis is being molded and welded. If the assembly line can initiate the process for building the engine and chassis at the same time, it could save precious resources.

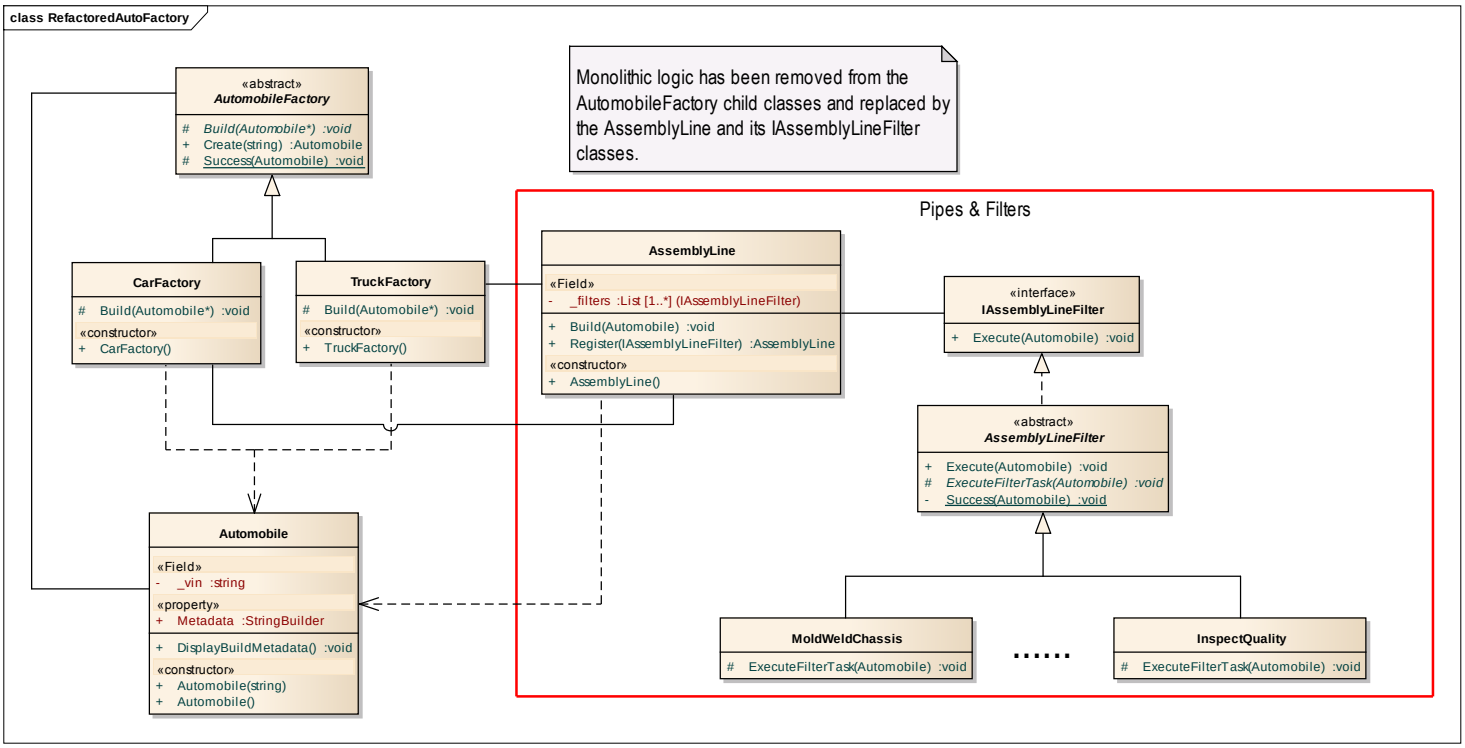

Figure 2-2: *Class diagram of refactored solution utilizing the Pipes & Filters pattern*

The class diagram illustrates the biggest difference between the two design solutions. The left half of the figure shows how there is not much of a difference between the original and refactored solution, but the right half highlights the main different; namely, the logical pipes and filters of the design solution. For the sake of conciseness, the concrete implementations of the filters are shown in abbreviated form denoted by the ellipses. However, the code for each filter component is provided below.

Automobile

No change from original.

AutomobileFactory

No change from original.

IAssemblyLineFilter

This interface provides a standardized contract for each filter component to implement. As each filter is encountered, its `Execute` method will be invoked by the pipeline (`AssemblyLine`) class.

```
public interface IAssemblyLineFilter
{
    void Execute(Automobile a);
}
```

AssemblyLine

This class is the logical representation of an automobile manufacturing assembly line. It serves as the "pipe" in the Pipes & Filters pattern. Filter components are registered with this class and when the `Build` method is invoked, the each filter's task is executed in sequential order. The return value of the `Register` method returns the acting `AssemblyLine` instance allowing for a fluent notation of filter component registration.

```
public class AssemblyLine
{
    private List<IAssemblyLineFilter> _filters = new List<IAssemblyLineFilter>();

    public AssemblyLine Register(IAssemblyLineFilter filter)
    {
        _filters.Add(filter);
        return this;
    }

    public void Build(Automobile a)
    {
        foreach (var filter in _filters)
        {
            filter.Execute(a);
        }
    }
}
```

AssemblyLineFilter

This class serves as the base class that all filter components will inherit from. This class ensures that as each filter component completes its task, that a status message is relayed to the console. Each child class will implement the `ExecuteFilterTask` method based on the specific requirements of its respective step in the build process.

```
public abstract class AssemblyLineFilter : IAssemblyLineFilter
{
    public void Execute(Automobile a)
    {
        ExecuteFilterTask(a);
        Success(a);
    }

    protected abstract void ExecuteFilterTask(Automobile a);

    private static void Success(Automobile a)
    {
        a.Metadata.AppendLine("SUCCESS!");
    }
}
```

Filter Classes (for cars and trucks)

These classes represent the "filter" components in the Pipes & Filters pattern. They are responsible for executing a single task within the manufacturing process for an `Automobile` object. The classes listed in this section are shared by the `CarFactory` and `TruckFactory` objects, but could just as easily be utilized in another factory class (such as a lawnmower or motorcycle factory).

Each class is a child of the `AssemblyLineFilter` class and therefore provides a specific implementation of the `ExecuteFilterTask` method. For the sake of this example, the specific implementation details for each filter component of the manufacturing process are symbolized by the ellipses.

```
public class MoldWeldChassis : AssemblyLineFilter
{
    protected override void ExecuteFilterTask(Automobile a)
    {
        a.Metadata.Append("Molding & welding chassis...");
        //...
    }
}

public class AddFrontSuspension : AssemblyLineFilter
{
    protected override void ExecuteFilterTask(Automobile a)
    {
        a.Metadata.Append("Adding front suspension...");
        //...
    }
}

public class AddRearSuspension : AssemblyLineFilter
{
    protected override void ExecuteFilterTask(Automobile a)
    {
        a.Metadata.Append("Adding rear suspension...");
        //...
    }
}

public class AddGasTank : AssemblyLineFilter
{
    protected override void ExecuteFilterTask(Automobile a)
    {
        a.Metadata.Append("Adding gas tank...");
        //...
    }
}

public class AddAxles : AssemblyLineFilter
{
    protected override void ExecuteFilterTask(Automobile a)
    {
        a.Metadata.Append("Adding rear axles...");
        //...
    }
}

public class AddDriveShafts : AssemblyLineFilter
{
    protected override void ExecuteFilterTask(Automobile a)
    {
        a.Metadata.Append("Adding drive shafts...");
        //...
    }
}
```

```
public class AddGearBoxes : AssemblyLineFilter
{
    protected override void ExecuteFilterTask(Automobile a)
    {
        a.Metadata.Append("Adding gear boxes...");
        //...
    }
}

public class AddSteeringBoxComponents : AssemblyLineFilter
{
    protected override void ExecuteFilterTask(Automobile a)
    {
        a.Metadata.Append("Adding steering box components...");
        //...
    }
}

public class AddWheelDrums : AssemblyLineFilter
{
    protected override void ExecuteFilterTask(Automobile a)
    {
        a.Metadata.Append("Adding wheel drums...");
        //...
    }
}

public class AddBrakes : AssemblyLineFilter
{
    protected override void ExecuteFilterTask(Automobile a)
    {
        a.Metadata.Append("Adding braking system...");
        //...
    }
}

public class MateEngineWithTransmission : AssemblyLineFilter
{
    protected override void ExecuteFilterTask(Automobile a)
    {
        a.Metadata.Append("Mating engine with transmission...");
        //...
    }
}

public class AttachRadiator : AssemblyLineFilter
{
    protected override void ExecuteFilterTask(Automobile a)
    {
        a.Metadata.Append("Attaching radiator...");
        //...
    }
}

public class MateBodyToFrame : AssemblyLineFilter
{
    protected override void ExecuteFilterTask(Automobile a)
    {
        a.Metadata.Append("Adding body to frame...");
        //...
    }
```

```
}

public class AddBatteryElements : AssemblyLineFilter
{
    protected override void ExecuteFilterTask(Automobile a)
    {
        a.Metadata.Append("Adding battery & spark plugs...");
        //...
    }
}

public class AddTires : AssemblyLineFilter
{
    protected override void ExecuteFilterTask(Automobile a)
    {
        a.Metadata.Append("Adding tires...");
        //...
    }
}

public class TopOffFluids : AssemblyLineFilter
{
    protected override void ExecuteFilterTask(Automobile a)
    {
        a.Metadata.Append("Topping-off fluids...");
        //...
    }
}

public class InspectQuality : AssemblyLineFilter
{
    protected override void ExecuteFilterTask(Automobile a)
    {
        a.Metadata.Append("Inspecting quality...");
        //...
    }
}
```

Filter Classes (for trucks only)

The following classes are more filter components, but these are specific to the build process for trucks. However, just as with the other filter component classes; these could also be utilized by other factories as desired.

```
public class AddTrailerHitch : AssemblyLineFilter
{
    protected override void ExecuteFilterTask(Automobile a)
    {
        a.Metadata.Append("Adding trailer hitch...");
        //...
    }
}

public class MateBedToFrame : AssemblyLineFilter
{
    protected override void ExecuteFilterTask(Automobile a)
    {
        a.Metadata.Append("Adding bed to frame...");
        //...
    }
}
```

CarFactory

This class is similar to its original version, but the biggest difference lies in its implementation details. The `Build` method is now solely responsible for initializing the `AssemblyLine` class (pipe) and the sequence of the build steps (filters) to be executed in order to build a car. The fluent notation provides an easy-to-read mechanism for specifying the execution order of the filter tasks. This also makes it easy to insert a new build step in the build process (as is seen in the `TruckFactory` class next). It should also be noted that the size of the class is greatly reduced (albeit at the cost of more classes in the overall solution, but there are other design patterns that can be utilized to minimize that impact as well).

```
public class CarFactory : AutomobileFactory
{
    protected override void Build(ref Automobile c)
    {
        var assemblyLine = new AssemblyLine();

                        // Build chassis
        assemblyLine.Register(new MoldWeldChassis())
                    .Register(new AddFrontSuspension())
                    .Register(new AddRearSuspension())
                    .Register(new AddGasTank())
                    .Register(new AddAxles())
                    .Register(new AddDriveShafts())
                    .Register(new AddGearBoxes())
                    .Register(new AddSteeringBoxComponents())
                    .Register(new AddWheelDrums())
                    .Register(new AddBrakes())
                        // Add engine
                    .Register(new MateEngineWithTransmission())
                    .Register(new AttachRadiator())
                        // Add body and finishings
                    .Register(new MateBodyToFrame())
                    .Register(new AddBatteryElements())
                    .Register(new AddTires())
                    .Register(new TopOffFluids())
                        // Quality Inspection
                    .Register(new InspectQuality())
                        // Invoke the build process.
                    .Build(c);
    }
}
```

TruckFactory

The `TruckFactory` class is obviously very similar to the `CarFactory` class. The key difference, however, is in the fact that this class uses extra steps in its build process (highlighted), but can easily reuse the existing filter classes that are used by the `CarFactory`, thereby removing duplicated code from the solution!

```
public class TruckFactory : AutomobileFactory
{
    protected override void Build(ref Automobile t)
    {
        var assemblyLine = new AssemblyLine();

                        // Build chassis
```

```
        assemblyLine.Register(new MoldWeldChassis())
                    .Register(new AddTrailerHitch())
                    .Register(new AddFrontSuspension())
                    .Register(new AddRearSuspension())
                    .Register(new AddGasTank())
                    .Register(new AddAxles())
                    .Register(new AddDriveShafts())
                    .Register(new AddGearBoxes())
                    .Register(new AddSteeringBoxComponents())
                    .Register(new AddWheelDrums())
                    .Register(new AddBrakes())
                        // Add engine
                    .Register(new MateEngineWithTransmission())
                    .Register(new AttachRadiator())
                        // Add body and finishings
                    .Register(new MateBodyToFrame())
                    .Register(new MateBedToFrame())
                    .Register(new AddBatteryElements())
                    .Register(new AddTires())
                    .Register(new TopOffFluids())
                        // Quality Inspection
                    .Register(new InspectQuality())
                        // Invoke the build process.
                    .Build(t);
    }
}
```

Program (Test Driver)

No change from original.

Output:

```
C-123ABC
===========
Molding & welding chassis...SUCCESS!
Adding front suspension...SUCCESS!
Adding rear suspension...SUCCESS!
Adding gas tank...SUCCESS!
Adding rear axles...SUCCESS!
Adding drive shafts...SUCCESS!
Adding gear boxes...SUCCESS!
Adding steering box components...SUCCESS!
Adding wheel drums...SUCCESS!
Adding braking system...SUCCESS!
Mating engine with transmission...SUCCESS!
Attaching radiator...SUCCESS!
Adding body to frame...SUCCESS!
Adding battery & spark plugs...SUCCESS!
Adding tires...SUCCESS!
Topping-off fluids...SUCCESS!
Inspecting quality...SUCCESS!

T-789XYZ
===========
Molding & welding chassis...SUCCESS!
```

```
Adding trailer hitch...SUCCESS!
Adding front suspension...SUCCESS!
Adding rear suspension...SUCCESS!
Adding gas tank...SUCCESS!
Adding rear axles...SUCCESS!
Adding drive shafts...SUCCESS!
Adding gear boxes...SUCCESS!
Adding steering box components...SUCCESS!
Adding wheel drums...SUCCESS!
Adding braking system...SUCCESS!
Mating engine with transmission...SUCCESS!
Attaching radiator...SUCCESS!
Adding body to frame...SUCCESS!
Adding bed to frame...SUCCESS!
Adding battery & spark plugs...SUCCESS!
Adding tires...SUCCESS!
Topping-off fluids...SUCCESS!
Inspecting quality...SUCCESS!
```

Non-Direct Activity Report

Date	Duration (minutes)	Specific Task / Activity
2-May-2015	115	Research/work on project #1.
3-May-2015	77	Research/work on project #1.
7-May-2015	147	Research/work on project #1.
11-May-2015	133	Research/work on project #1.
12-May-2015	180	Research/work on project #1.
13-May-2015	210	Research/work on project #1.
17-May-2015	45	Research/work on project #1.
19-May-2015	308	Research/work on project #1.
21-May-2015	124	Research/work on project #1.
22-May-2015	231	Research/work on project #1.
Sum for Report #1	**1570**	**/ 900 (1 week @ 900/wk)**
Sum for Report #2		**/ 900 (1 week @ 900/wk)**
Sum for Report #3		**/ 900 (1 week @ 900/wk)**
Sum for Report #4		**/ 1800 (2 weeks @ 900/wk)**
Sum for Class	**1570**	**/ 4500 (5 weeks @ 900/wk)**