The Facade Design Pattern

Graeme Geldenhuys

2011-12-26

The Facade Design Pattern describes a way of making a subsystem (a set of classes that work closely together) easier to use. The Facade allows us to simplify the interface to such a subsystem with a higher-level interface. It also promotes the idea of avoiding tight dependencies on the components of such a subsystem. This article will introduce the Facade Pattern using an easy to understand example, and also touch on another design pattern (the Adapter Pattern) that sounds very similar in behaviour to the Facade - but we will highlight some important differences between them.

Stop the bus! What is a Design Pattern?

Wikipedia describes a software design pattern as follows[1]:

"*In software engineering, a design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. A design pattern is not a finished design that can be transformed directly into code. It is a well documented description of how to solve a problem that can be used in many different situations.*"

In short, design patterns describes a solution to a design problem that some clever people already solved. A very important point to make though, is that design patterns are NOT code templates!

Design Patterns are very useful to software designers. They can speed up the development process by providing tested and proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns helps to prevent subtle issues that can cause major problems later. The way Design Patterns are documented, they also use consistent terminology – thus communicating a design solution, using design pattern terminology, to another programmer that is also versed in design patterns, makes them understand the design solution much better. It also improves code readability for coders and architects familiar with the patterns.

1 Wikipedia (English Language): Software Design Pattern (http://en.wikipedia.org/wiki/Software_design_pattern)

Design Patterns are normally classified into one of three groups – based on their purpose.

- *Creational*: patterns that work towards the process of creating objects.
- *Structural*: patterns that deal with the composition of classes or objects.
- *Behavioural*: patterns that affect the ways in which classes or objects interact with each other and distribute responsibility.

The Facade Pattern is classified as a Structural design pattern.

For more in-depth information about Design Patterns, please visit your local book store, or simply search the Internet. There is a lot of information about design patterns all over the Internet.

Back to our problem example

Lets start off by jumping straight into our problem example - highlighting what the problem is with our example, and how the Facade Pattern can help us solve the problem, or at least greatly improve the design.

Something most of us have probably wished for, or tried to build some way or another, is a Home Theatre System. An awesome system that can play DVD's with surround sound speakers, a wide-screen projection system, and even a popcorn machine. What better way to relax, put your feet up and listen to some music or watch your favourite blockbuster movie.

Now lets assume you just completed all the hard work by finally installing all the various parts of your system – the amplifier, projector, projector screen, speakers etc. All that remain is for us to switch everything on and watch a movie in its full glory! But what exactly are the steps to switch on your home theatre system and watch that movie? Here is a list of things we need to do, in a specific order, just so we can watch our favourite movie.

1. Load the Popcorn Machine
2. Switch on the Popcorn Machine
3. Lower the Projector Screen
4. Switch on the amplifier
5. Set the amplifier volume to medium
6. Set the amplifier input to DVD
7. Switch on the DVD player
8. Load the DVD
9. Set the DVD player to surround sound mode
10. Switch on the projector
11. Set the projector input to DVD
12. Set the projector to wide-screen mode

13. Dim the lights
14. Start playing the DVD

What a list! We have to fiddle with so many different components, apply the correct settings on multiple components etc. Makes one wonder... Do you have to do this every time? How complex would the process be if I wanted to switch everything off once we are done watching our movie? Do we have to follow that whole list of steps again, but in reverse? What if we wanted to listen to some music instead of watching a movie? Would we end up with such a long list of step too?

A long list of questions, and to most of them the answer is the same: Yes, we would require another long list of steps to follow. Sigh.

To help visualise our home theatre system in terms of a software project, I have created a high-level component diagram representing our home theatre system, and indicated using arrows, how each component might depend or interact with other components. The diagram also indicates how the client [that is us] interacts with each component.

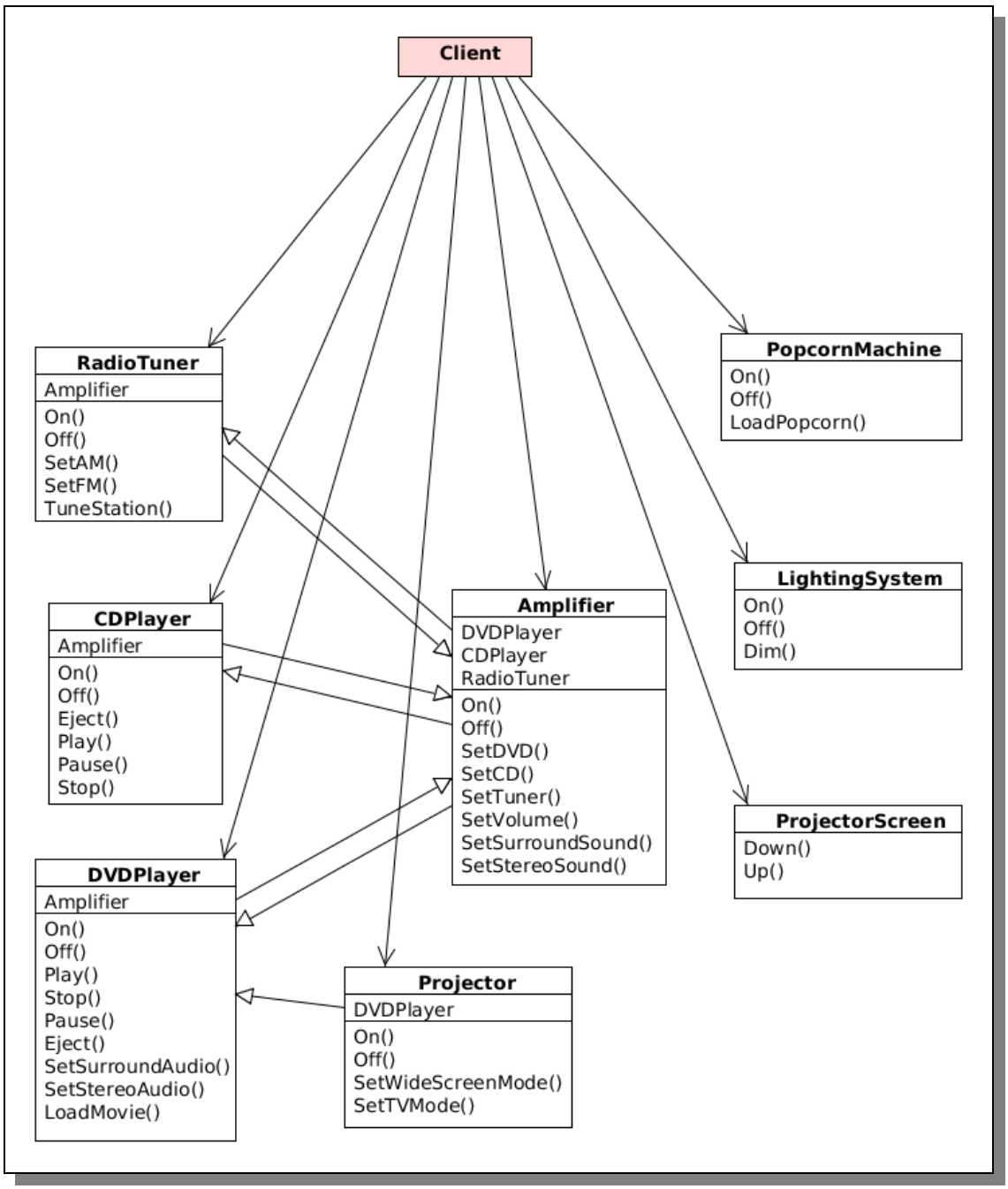

Figure 1: Home Theatre System showing component interactions.

Now if we had to implement our home theatre system in terms of software and classes, this is how our list of steps would look in terms of source code:

```
Listing 1:
PopcornMachine.LoadPopcorn;
PopcornMachine.On;

ProjectorScreen.Down;

Amplifier.On;
Amplifier.SetVolume(50%);
Amplifier.SetDVD(DVDPlayer);
```

```
DVDPlayer.On;
DVDPlayer.LoadMovie(MovieName);
DVDPlayer.SetSurroundAudio;

Projector.On;
Projector.SetWideScreenMode;

LightingSystem.On;
LightingSystem.Dim;

DVDPlayer.Play;
```

Those are a lot of objects and a lot of different interfaces that we need to learn before we can enjoy our home theatre system.

And to make matters worse, the whole system can become even more complex over time too. What if we wanted to add a Blu-ray Disc Player to our home theatre system? Or we wanted to start watching television via our newly purchased satellite decoder?

Facade to the rescue

Even though design patterns might sound complex at first, the Facade Pattern is actually very easy to understand and implement. The Gang-of-Four book (which is often considered the bible of design pattern books) has the following official explanation of what the intent is of the Facade Pattern[2]:

"*The Facade Pattern provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.*"

That doesn't sound too complex at all – so lets follow what it says. First we introduce a new facade class into our home theatre system. Our facade class will treat our home theatre system like a subsystem – many components that work together. Please see Figure 2 on page 6 for an overall picture of how we changed the home theatre system's design.

The new class we introduced is called `HomeTheatreFacade`, which introduces new methods for us [the client] to use. The `HomeTheatreFacade` class talks directly to the Home Theatre subsystem. We then modify the client code to talk to the `HomeTheatreFacade` class, instead of talking directly to the various subsystem components.

The new facade class greatly simplifies the usage of the subsystem. Now, instead of having to following a thirteen step process to watch a movie, we can simply call `HomeTheatreFacade.WatchMovie()`. How easy is that! What really makes the Facade Pattern even better, is that it doesn't hide the subsystem components from us [more about this later], it just makes them easier to use. If we need to access the low-level functionality of a certain component in the subsystem, we can still do so without limitations.

2 Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley 1994. ISBN 0-201-63361-2 (page 185).

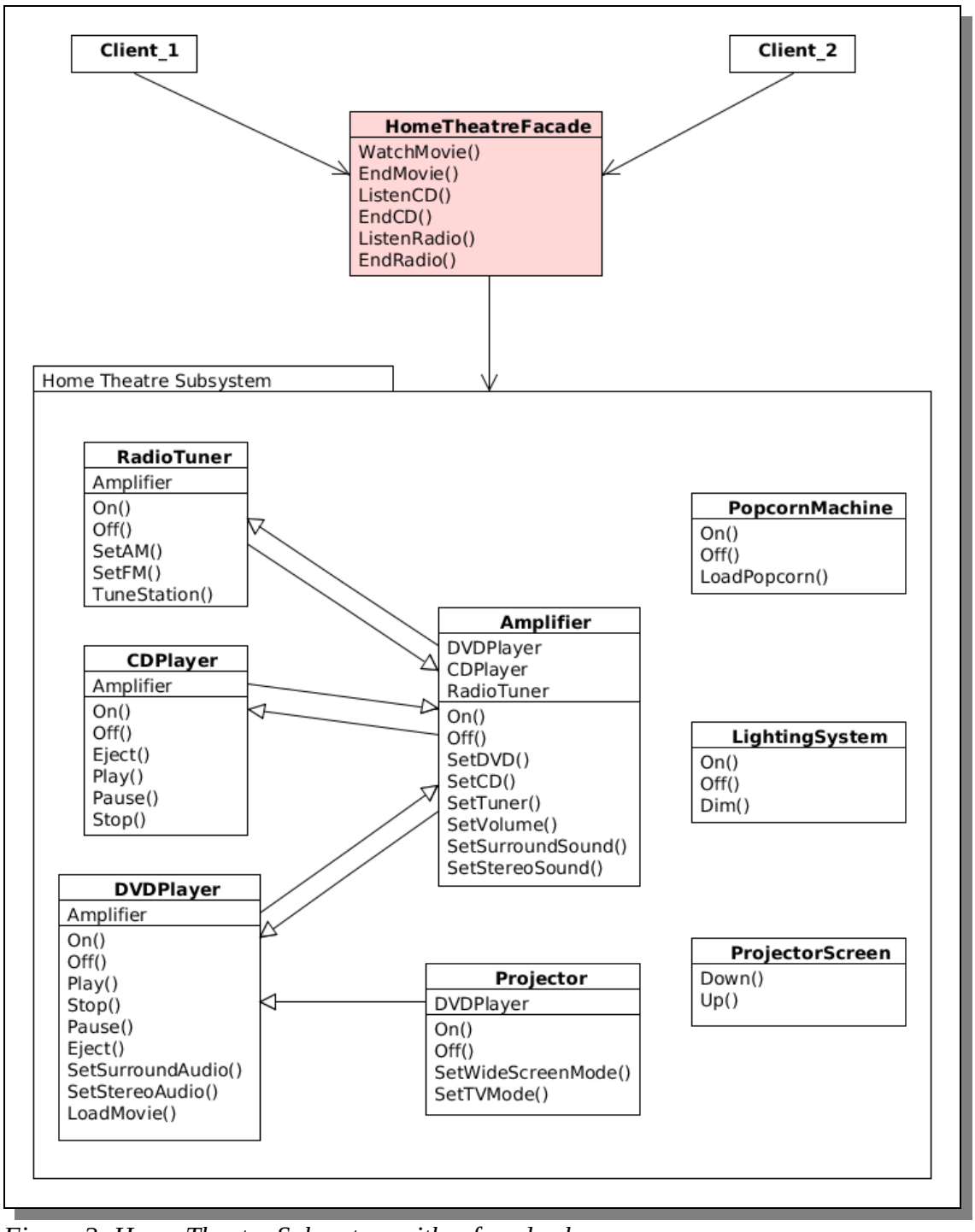

Figure 2: Home Theatre Subsystem with a facade class.

Home Theatre Facade implementation

So lets cover the details of how to implement the `HomeTheatreFacade` class, and how to use it. A facade implementation often takes advantage of *composition* – one of many good object oriented principles to follow. In short, *composition* is when one class keeps instances or references of other objects in private field variables, so they can be referred to later when needed. *Composition* is a object oriented principle often preferred over

Inheritance.

Here is the class interface definition for our `HomeTheatreFacade` class.

```
  THomeTheatreFacade = class(TObject)
  private
    { This is the composition I mentioned }
    FPopcorn: TPopcornMachine;
    FLights: TLightingSystem;
    FScreen: TProjectorScreen;
    FAmp: TAmplifier;
    FDVD: TDVDPlayer;
    FCD: TCDPlayer;
    FTuner: TRadioTuner;
    FProjector: TProjector;
  public
    constructor Create(APopCorn: TPopcornMachine;
        ALights: TLightingSystem;
        AScreen: TProjectorScreen;
        AAmplifier: TAmplifier;
        ADVDPlayer: TDVDPlayer;
        ACDPlayer: TCDPlayer;
        ATuner: TRadioTuner;
        AProjector: TProjector);
    procedure WatchMovie(const AMovieTitle: string);
    procedure EndMovie;
    procedure ListenCD(const ACDTitle: string);
    procedure EndCD;
    procedure ListenRadio(const AFrequency: double);
    procedure EndRadio;
  end;
```

The first step in implementing this class, would be to construct an instance of the `THomeTheatreFacade` and make sure it uses the preferred OO principal, Composition, to keep track of the subsystem components. To do this, we simply pass the component instances of the Home Theatre subsystem to the `THomeTheatreFacade`'s constructor. Here follows the source code implementation of the constructor:

```
constructor THomeTheatreFacade.Create(
    APopCorn: TPopcornMachine;
    ALights: TLightingSystem;
    AScreen: TProjectorScreen;
    AAmplifier: TAmplifier;
    ADVDPlayer: TDVDPlayer;
    ACDPlayer: TCDPlayer;
    ATuner: TRadioTuner;
    AProjector: TProjector);
begin
  { Here we assign a reference to each component of the
    subsystem to private field variables }
  FPopcorn := APopcorn;
  FLights := ALights;
  FScreen := AScreen;
  FAmp := AAmplifier;
  FDVD := ADVDPlayer;
  FCD := ACDPlayer;
  FTuner := ATuner;
```

```
  FProjector := AProjector;

  { do other processing here if needed }
end;
```

Now our facade class knows about all the subsystem components to do its job. So lets hide the complex usage of the home theatre subsystem behind a single method - called the `WatchMovie()` method. Remember, the goal of the Facade Pattern is to simplify the interface to a subsystem. Once implemented, all we need to do to watch a movie, is pass in the title of the movie we want to watch. Internally the `WatchMovie()` method communicates with all the required components of the home theatre system - completing that long thirteen step process we listed earlier. Here follows the source code implementation of `WatchMovie()`:

```
procedure THomeTheatreFacade.WatchMovie(const AMovieTitle:
    string);
begin
  FPopcorn.LoadPopcorn;
  FPopcorn.On;

  FScreen.Down;

  FAmp.On;
  FAmp.SetVolume(50);
  FAmp.SetDVD(FDVD);

  FDVD.On;
  FDVD.LoadMovie(AMovieTitle);
  FDVD.SetSurroundSound;

  FProjector.On;
  FProjector.SetWideScreenMode;

  FLights.On;
  FLights.Dim(20);

  FDVD.Play;
end;
```

The last part of our simplified subsystem interface we need to implement, is the `EndMovie()` method. This will contain the steps to switch off our home theatre system once we are done watching our movie. The facade class will once again take care of all the complex communications with each subsystem component. Here follows the source code implementation for the `EndMovie()` method:

```
procedure THomeTheatreFacade.EndMovie;
begin
  FLights.Dim(100);

  FDVD.Stop;

  FPopcorn.Off;

  FProjector.Off;

  FDVD.Eject;
  FDVD.Off;

  FAmp.Off;

  FScreen.Up;
end;
```

Finally, all the hard work is done! Our facade class is now complete, and will allow us to play and stop movies with ease. What remains now, is for us to change our client code to rather use the new home theatre facade class, instead of talking directly to the various complex subsystem components. Here is the code to show how it is done:

```
{ Watching a movie with the help of the Facade class }
var
  HomeTheatre: THomeTheatreFacade;
begin
  { ...instantiate the home theatre components here... }

  HomeTheatre := THomeTheatreFacade.Create(Popcorn, Lights,
              Screen, Amp, DVD, CD, Tuner, Projector);
  try
    HomeTheatre.WatchMovie('The Bourne Identity');
    HomeTheatre.EndMovie;
  finally
    HomeTheatre.Free;
  end;

  { ...free the home theatre components here... }
end.
```

The full source code of this example project is included on the cover DVD of the FreeX magazine. It is written in Object Pascal and can be compiled with the Free Pascal Compiler[3]. Figure 3 is a screen-capture showing the runtime output of the completed home theatre project.

3 The Free Pascal Compiler (http://www.freepascal.org)

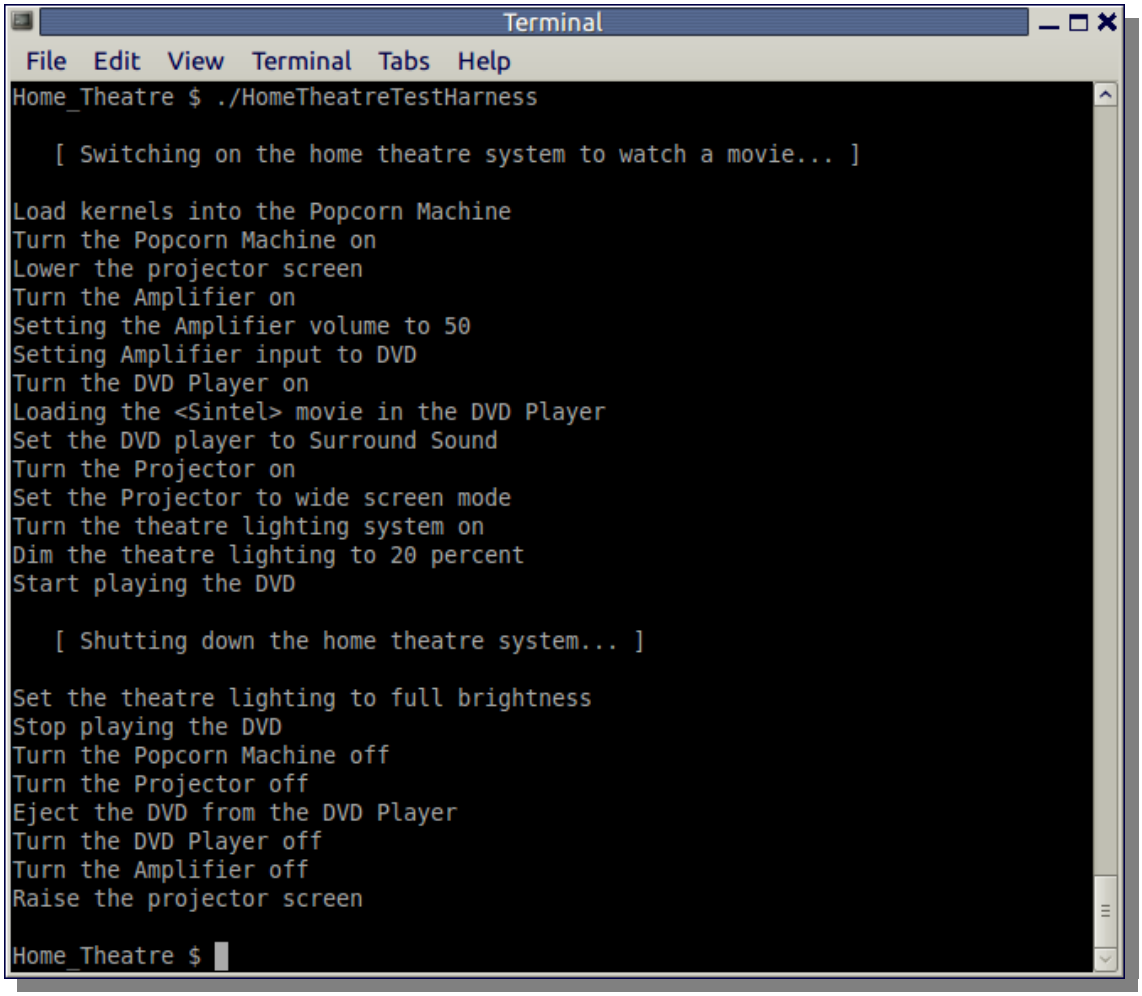

Figure 3: Output of the completed example project.

Final Thoughts

I often get asked questions regarding the similarity between the Facade Pattern and the Adapter Pattern[4]. From a distance it might look like they are doing the same thing, but they are not. As you now know, the Facade Pattern simplifies the interface of a subsystem. The Adapter Pattern on the other hand, changes the interface of one or more classes, to what a client expects. A real-world example of the Adapter Pattern can easily be described with the help of a AC power/travel adapter.

For example: a US tourist would have to get a AC power adapter to change the American AC plug on his laptop, so that it could fit into a European wall outlet. In terms of software design patterns, the American AC plug on the laptop is the *client*, the European wall outlet is the *target interface* the tourist expects his laptop to work with, and the AC power adapter is the *adaptor class* doing the interface changing.

The confusion between the two patterns often start by the developer thinking that it is about how many classes get “wrapped” by the Facade or Adapter pattern. This is wrong, it is rather about their *intent*. The intent of the Adapter pattern is to *alter* an interface,

4 Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley 1994. ISBN 0-201-63361-2 (page 139).

whereas the intent of the Facade is to provide a *simplified* interface.

The other misconception is that the facade class is not limited to only passing through requests from the client to the subsystem. The facade class is free to add extra functionality, and make it a more “powerful” interface class in need be.

Similarly, the subsystem is not limited to only one facade class. You might have a very complex subsystems, which could very well benefit from having more than one facade class. Different clients could then make requests to different facade classes – depending on what the client wants done. One facade class could even make requests to another facade class.

This brings me to the end of my article. I hope you found this information informative, and that you now have the knowledge to use the Facade Pattern [and hopefully other design patterns too] in your projects. If nothing else, this is one extra tool in your “programmers toolbox”!