Chapter 1

Design

Karl Fogel and Mosha Bar observes, in chapter 8 of their book on CVS and Open Source development [**?**], that good design in open source projects is usually attained by having a well designed core and architecture and then allow the rest of the design to incrementally grow from this.

Detailed design decisions should be postponed as long as possible so that the participants have time to learn as much as possible about the software before making them. We have tried to do this by only designing the architecture (section 1.2) as well as the domain datastructure(section 1.3.1) up front. Designs on lower levels such as the design of the undo functionality, the sound system design and many other, lesser, designs have been done right before the code for them has been implemented.

This way our design have incrementally grown from some initial "written-in-stone" decisions such as: "We will have an architecture based on the observer pattern" and "We will have a Facade for communicating with the Domain".

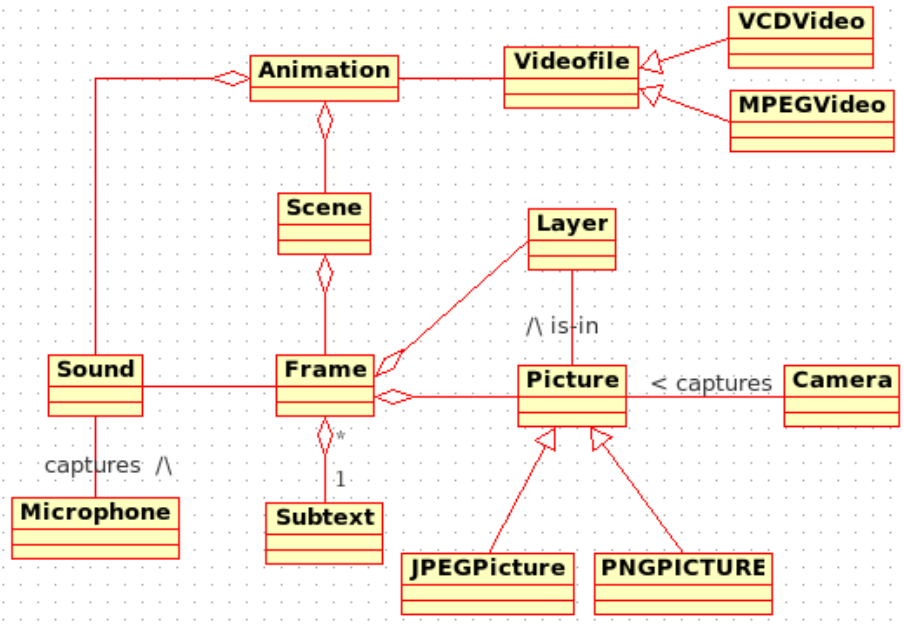

Figure 1.1: Domain model

1.1 Domain

When creating an application one is trying to solve a problem. This problem lies in some problem domain and it is useful to analyze this domain before designing the application.

To do this we spent a lot of time in the beginning on identifying domain entities and how these were logically connected. We then summed up what we had learned in a Domain class diagram which is shown in figure 1.1. This model is quite exhaustive and not all of the concepts in it were found to be useful once we learned more about the problems our program is meant to solve.

Parts of the Domain Model then served as the inspiration for the design of the domain layer. The rationale behind this is to make the application design as close to the problem as possible which makes it easier to comprehend and to talk about the design.

1.2 Architecture

The architecture we have chosen is a five layer architecture based on the “loose layers” principle [**?**, **?**]. Figure 1.2 shows the layers in Stopmotion with arrows describing how the layers communicate (which ones are coupled). The reason for the layers is to split the different services into higher level services, such as those in the presentation layer where the GUI code are, and lower level services such as adding of frames to the model

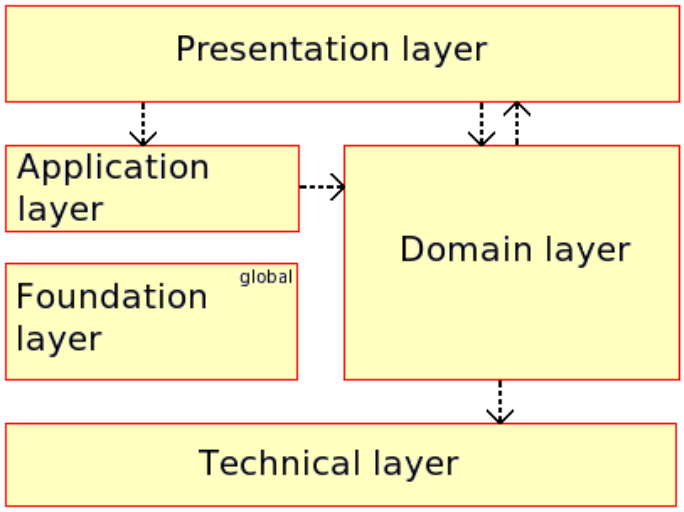

Figure 1.2: Communication between layers

(Domain layer) and exporting to video (Technical layer).

The Application layer acts as a mediator between the Presentation and the lower layers, forwarding requests from the Presentation layer. The final layer; the foundation layer, is a utility layer consisting of global singletons which offer services such as logging and preferences saving which can be used by all the other layers except from the technical which is completely self sufficient.

The end user will only interact with the presentation layer which in turn will ask for services from lower layers.

1.2.1 Domain Facade

To control the access to the domain layer we chose to implement a *singleton*[**?**] class called the DomainFacade (see figure 1.3). The Domain Facade takes care of initializing all the required classes in the Domain as they are needed.

Making the Domain Facade a singleton has one repercussion. It means the application can only handle one animation at a time. We are aware of this but we don't think multiple animations in the same program is something this application should have. This feature will complicate the user interface, and we doubt many of the target users will need it. If they with to work with several animations at the same time they can open several programs and move things between the animations by drag-and-drop.

If multiple animations should become desirable at a later stage it won't be too hard to subclass the facade and make a new class handling multiple animation domains. This will only require changing code in the presentation and application layer.

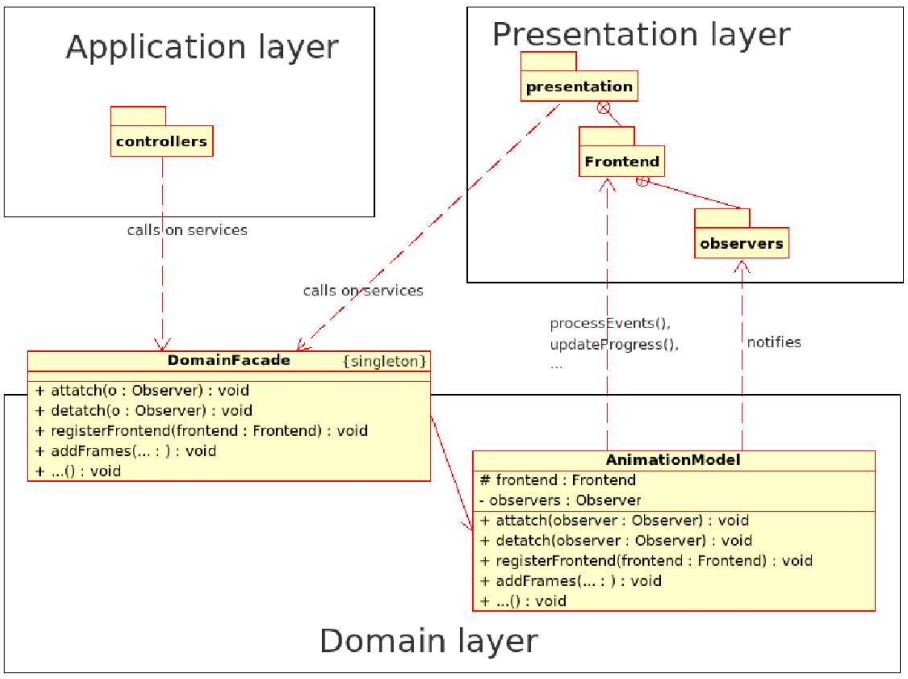

Figure 1.3: The frontend, observer and facade architecture

1.2.2 Frontend packages

As explained in the supplementary specification in section **??** on page **??** one of the requirements for Stopmotion was that it should be easy to change the GUI of the application without changing many parts of the code.

For this reason we chose to split the presentation layer into several frontend packages. A frontend is set of classes enabling the use of the application, but it doesn't necessarily have to be GUI based. As the frontends lies in the presentation layer they are the part of the program the users will communicate with. The frontends in turn calls on services from lower layers which performs the application specific services and operations.

A frontend package must at least have a frontend class for starting the application and can also have one or more observers for displaying the data in the data model. GUI frontends will also have general GUI code for displaying windows, receiving events from the user, etc. Figure 1.4 shows the Frontend class and the Frontend sub-classes for the current frontends.

Both the observers and the frontend class are registered dynamically through function calls and this way one can easily write new frontends, replace existing ones or add new observers with new ways of displaying the data without changing a single line of existing code. One example of a new Observer could for example be a list showing the scenes.

All of this means that even though our application is largely written using Qt, this is only one frontend and we can both compile and link the application without this library using the NonGUI frontend. In the future someone might also wish to write a GTK GUI frontend to ship the application without Qt, or they might want to port it to Windows or Macintosh. With the frontend design this can be done without the pains of having to port all the layers.

In addition to allow for the addition of new frontends and views this structure makes our program very modular and thereby very predictable. It is quite painless to modify the software as it is easy to guess where in the source code some functionality can be located.

Communication with the frontend class

Because we're using a layered architecture with downward communication we cannot communicate with the user directly from the lower layers. In many cases it's important to tell the user that something went wrong and what he/she can to do to avoid the error. We also need the ability to tell the frontend to process its events on time-consuming operations so that the GUI doesn't lag and to tell it when to update the progress-bar when doing these operations. As explained above we chose to do this communication by registering the frontend dynamically at runtime.

It's only possible to run one frontend at the same time, but it's no problem to e.g. implement and use a GTK frontend for processing events. It means that we can have different frontends with different implementations for doing the same thing. This is possible because we have defined an interface in the form of the class Frontend so that the API for the Domain layer (see 1.3 on page) are the same no matter which frontend

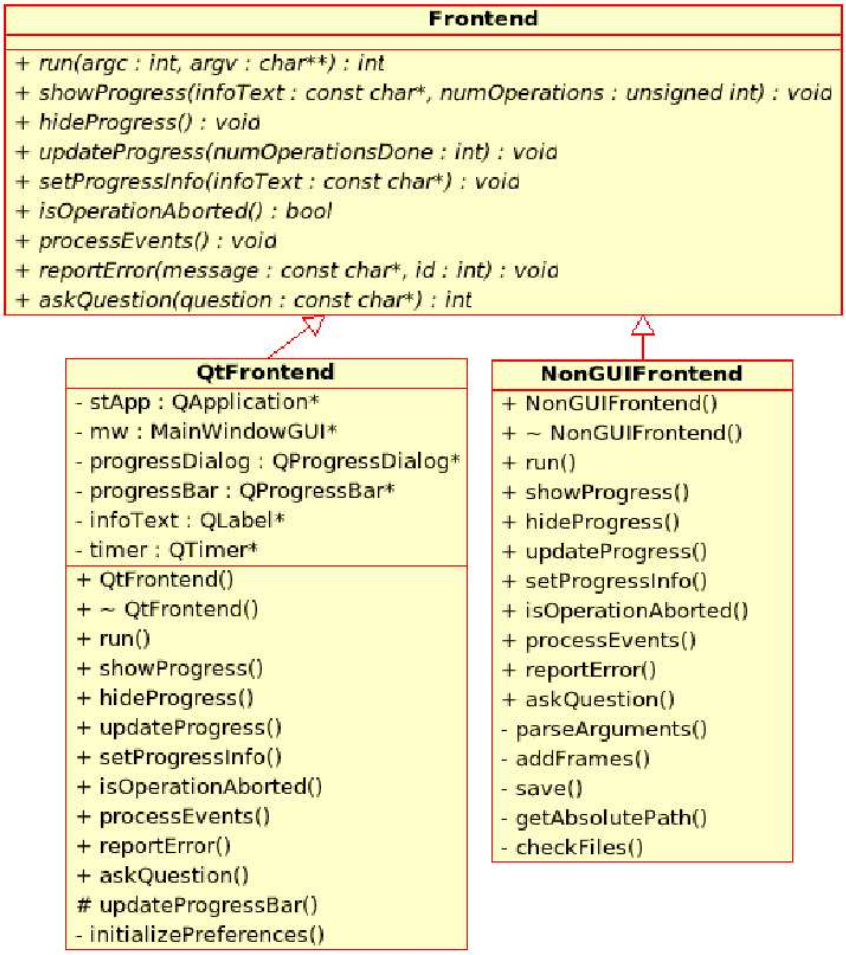

Figure 1.4: The Frontend class and the present frontends (simplified)

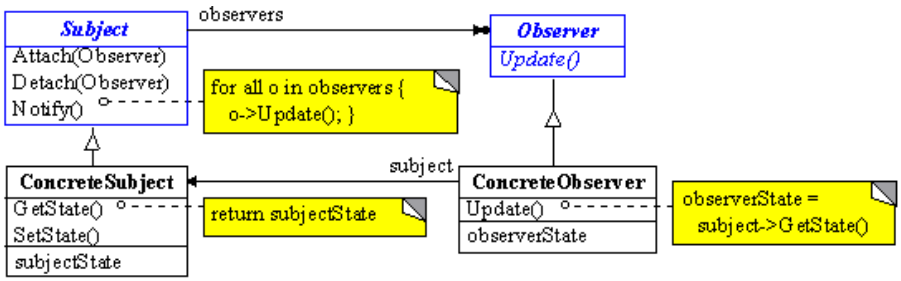

Figure 1.5: The Observer pattern[**?**]

are being used.

All of the frontends have to extend this class and then they can have their own implementation for doing the error reporting, processing of events, displaying a progress bar, etc.

1.2.3 Observers

As mentioned above most frontends has one or more observers for displaying the data in the Domain data structure, such as the FrameView and the FrameBar classes. For information distribution when this model changes we have chosen an approach based on the observer design pattern based with the push model[**?**]. Figure 1.5 shows the classical sketch of the observer pattern with the pull model. Our implementation with the push model is almost the same except from that our notify functions sends information about the change so that the update functions don't have to query the *ConcreteSubject* for the new state.

As mentioned the observers are registered in the Domain at runtime and when the Domain receive a request which changes the data model it notifies the observers which in turn can take appropriate steps to display these changes to the user.

This is a very flexible way of spreading changes through the system and have lead to an architecture where the model and the classes dependent on the model (the observers) are clearly separated. This makes it very easy to change one without affecting the other and we can add new observers, for instance a new way of displaying the animation in slow motion, easily.

The push model makes this model a little less flexible, but we feel the extra performance benefits this model gives us greatly outweigh the disadvantages.

1.3 Designs

1.3.1 Animation model

The animation data model consists of two parts, an interface and an implementation. The abstract class AnimationModel is the interface to the model and the implementation of the model has to conform to this interface by inheriting it. The only functionality

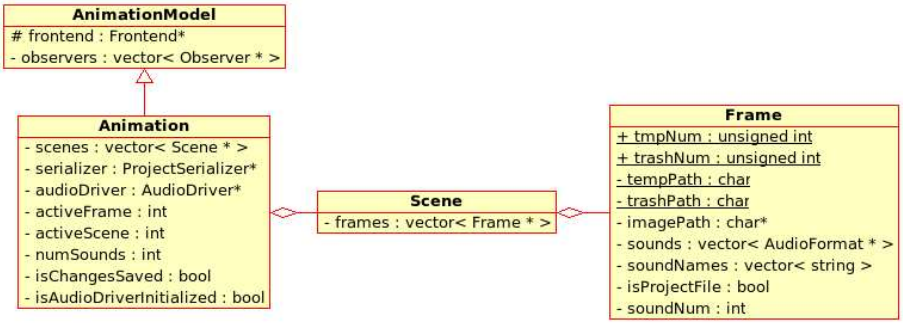

Figure 1.6: Animation Model

in this class is to notify the observers of a change in the model and thus it fills the role of the *Subject* in the Observer pattern.

The implementation of the model consist of three classes: Animation, Scene and Frame. As visualized in figure 1.6 the Animation class contains some information related to the animation as a whole as well as the scenes. The Scene class in turn contains the frames and the frames contains the information for one frame in the animation such as the path to where the image for this frame is, sounds linked to it, etc. The Animation with its underlying classes are the *ConcreteSubject* in the Observer pattern.

By separating the interface to the animation model from the implementation we can later change the implementation of the model without affecting the classes which use it and this further increases the modularity of the software.

1.3.2 Undo and redo functionality

The undo and redo functionality is designed as an adaption of the Command pattern[**?**]. Every command made to the Domain is stored as an Undo command object and these are linked together in an UndoHistory class as seen in figure 1.7.When the user wants to undo an command the UndoHistory will find the appropriate undo object and use it to undo the operation by feeding it with the animation model.

There are six different undo objects, one for each undoable operation. All inherits from the Undo class and the constructors for each takes the information they need to undo their type of command. The commands are undone by calling the reverse command on the animation model. Thus an `UndoRemove->undo(...)` would lead to a call to the `addFrames(...)` function in the model.

One example of the constructor API of an undo object is:

```
UndoRemove (const vector<char *> &frameNames,
      unsigned int fromIndex, int activeScene);
```

This information is all that is needed to undo the remove operations and ensures that the undo information is very compact which means we can store allot of undo objects.

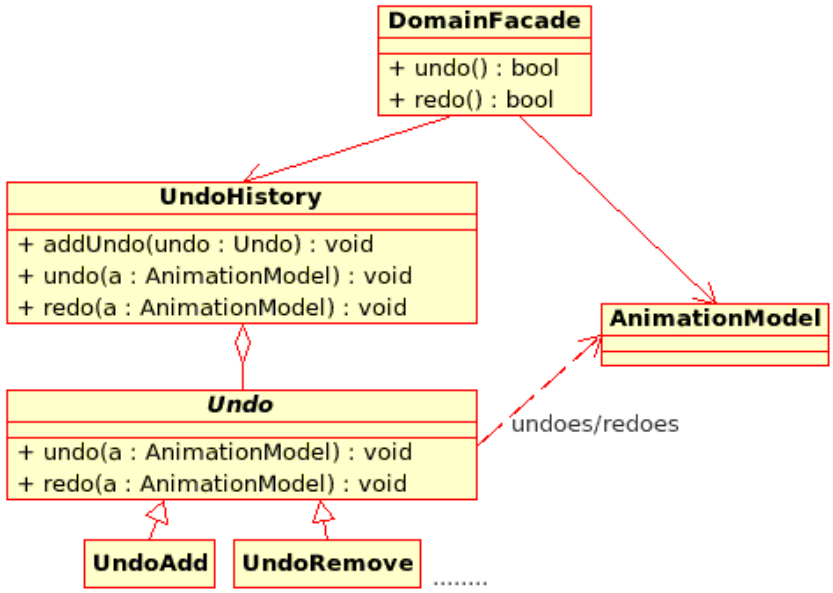

Figure 1.7: The Undo design

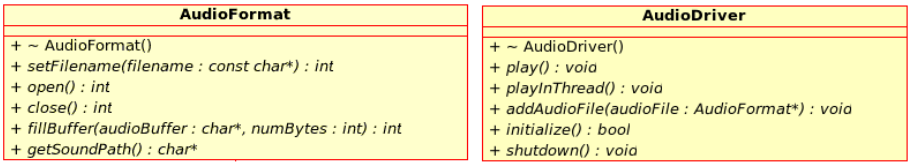

Figure 1.8: The Audio interfaces

One alternative to the command pattern adaption of the undo functionality is to have a design based on the Memento pattern. With this design we would have stored the state of the animation between each operation. This way one could have undone commands by switching to a previous state and although we considered this approach early on, it would have cause allot of information to be stored in the memory and thus wouldn't have been very efficient.

1.3.3 Audio formats

Nowadays there are many audio formats used by people, and there is no reason for thinking that we shouldn't get more of them in the future. It's therefore important to have a general interface for implementing different audio formats such as mp3, waw, ogg ... you name it! This makes it easier to implement audio drivers which are capable of playing all the different formats. The AudioFormat interface in Stopmotion is depicted in figure 1.8 and all the different format classes must inherit from this.

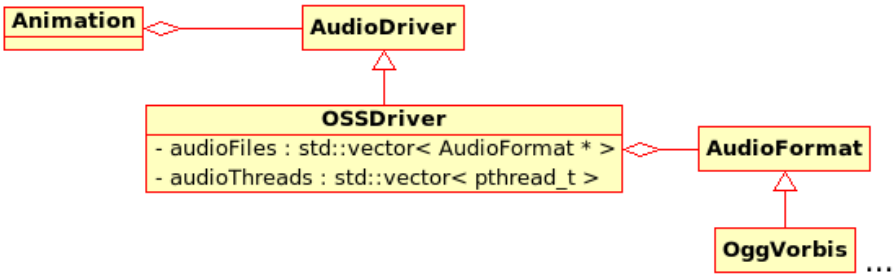

Figure 1.9: The Audio design

When we want to play a format we just attach it to an audio driver class and let it communicate with this. Since all the format classes has an uniform interface it is easy to make general audio drivers which can play any format.

1.3.4 Audio drivers

The audio drivers handles the initialization and shutdown of the audio device, as well as the playing of sounds. There is also a general interface for the audio drivers which is also shown in figure 1.8. The driver is coupled to the model which sends it requests for initializing, shutdown and playing of sounds. The driver in turn communicates with an AudioFormat class which it use to fill a buffer with data which is then flushed directly to the audio device.

How these classes are couplet together are depicted in figure 1.9.

Having a general interface for the audio driver classes makes it easy to implement different drivers such as ALSA and OSS without having to make changes in the model. This is handy if we want to port the application to an another platform or the current audio driver becomes deprecated.