# Network: A Pattern for Composing Computation


## Abstract:

In Oticons software product for fitting hearing aids OtiSet there's a need to simulate the digital hearing aids performance under different conditions (traffic noise, loud speech etc.). Simulating the hearing aids implies simulation of discrete components organized in a network through which the representation of sound i.e. the signal propagates. Each component performs a computation on the signal e.g. filtering (of frequencies), Volume Control (amplification) etc. It's desirable if the software were capable of reusing the components.

I've looked for patterns that could be used because I was sure I wasn't the first to deal with this problem of composing computation. The relevant patterns that I came across (see reference) didn't have any code examples, and tended to be high-level, vague, describing and categorizing using a lot of options.
In this paper I've decided to describe a pattern with coding examples, and have chosen some of the options in the hope that it will be easier for people to build something based on this more concrete pattern description.


## Intent

Describe a computation by a network of black-box components. Data streams through the network, with each component taking data from its inputs and delivering it to its outputs. The network connects output from one component to the input of another.

## Also Known As

Pipes and Filters, Pipeline

## Motivation

Computation is often organized as a network of components taking input and delivering output without changing their own state. An example of computations organized as a network is the simulation of Digital Signal Processing (DSP). DSP is concerned with the representation of signals by sequences of numbers or symbols and the processing of these sequences.
An example of a DSP is the chip in a modern programmable digital hearing aid which transforms the input sound into a digital signal, processes the signal and transforming the resulting digital signal into sound again providing better hearing for the hearing impaired person.
The chip consists of various discrete components that are connected to each other. A simplified example is shown in Figure 1 where each box represents a discrete manipulation or computation. Information flow between two boxes is called a *signal*. Signals can branch out to different components as shown with the output from the **`VolumeControl`** and a single component can gather several signals as shown with the **`Adder`**.

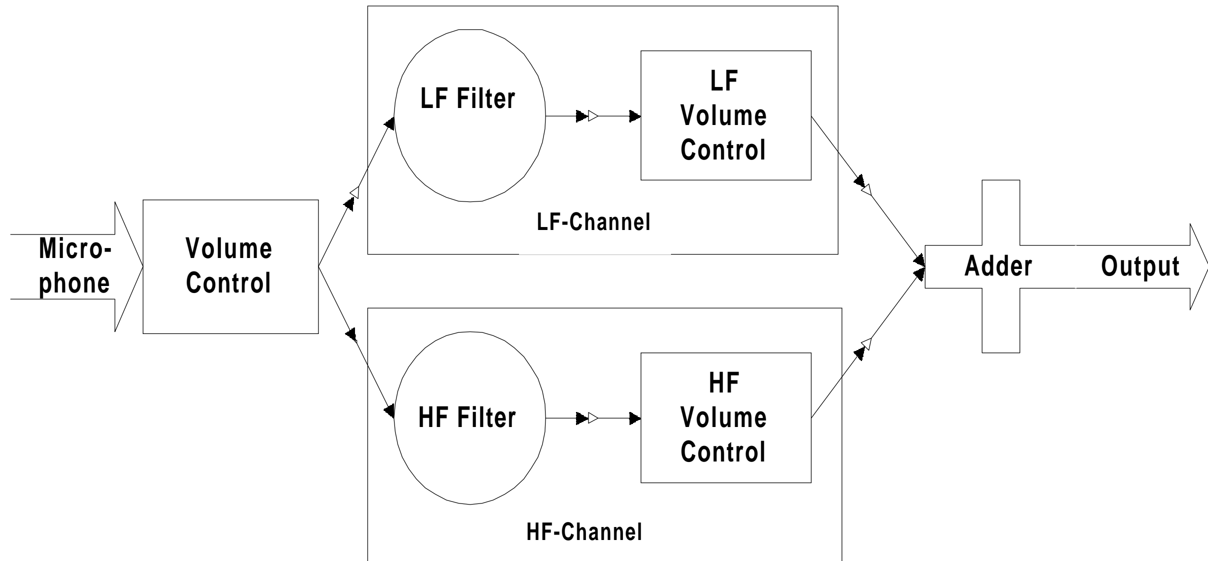


Figure 1, A DSP network

In general, a network computes a function based on its components. For example, the output of Figure 1, can be described in a Lisp-like way as:

```
Output = Adder( LFChannel( VolumeControl( Microphone( ))),
                HFChannel( VolumeControl( Microphone( ))))
```

Equation 1, Output relation

In Equation 1 the outermost function is **Adder** then **LFChannel** and **HFChannel** which indicates that computation uses some kind of backwards propagation. The Network pattern should do the same, meaning that **Adder** asks for a result in **LFChannel** and **HFChannel** and so forth. This requires that the output of **LFChannel** and **HFChannel** be unified with the input to **Adder**.

Networks support hierarchical decomposition. The **LFChannel** (LF = Low Frequency) and the **HFChannel** (HF = High Frequency) are composite components containing an internal network. The internal network or sub-network is composed of a filter and a volume control and the relation of Equation 2 applies.

```
LFChannel( input ) = LFVolumeControl( LFFilter( input ))
HFChannel( input ) = HFVolumeControl( HFFilter( input ))
```

Equation 2, Sub-network relations

Instead of having a mediator handling everything centrally, the Network pattern distributes responsibility of flow to each class. Each component knows the components that produce the signals it uses, but not the ones that use the signals it produces. The network produces output by having **Adder** ask **LFChannel** and **HFChannel** for output, and they in turn ask **VolumeControl** which they both depend upon. The **VolumeControl** ask the **Microphone** for output before computing and delivering its own output to **LFChannel** and **HFChannel**.

The network must not be cyclic, thus the network can be seen as a set of interconnected components organized in a Directed Acyclic Graph (DAG).

The information flow or signal between the components is a separate class hierarchy that allow for different kinds of information to flow between the components. Each Component derived class (see Figure 2) has its own Signal that is the output from that class.

The network class (see Figure 2), which acts as a kind of facade (see ref [Gam95], facade pattern), is responsible for the high-level interface to the network, including construction and destruction of components, and initialization of the input before querying the output.

The network class must know where to place output and where the input can be fetched. Components that provide the input are called *starters* and the components that provide the output are denoted *ends*. A network pattern implementation can start and end in several components. However, all starter components on which the end components depend must have been initialized.

## Applicability

Use the Network pattern

- to perform a computation consisting of smaller interconnected parts
- to enable reuse of computational parts by sharing an interface
- to distribute responsibility of the flow
- to allow for dynamic connectivity of a computation

## Structure

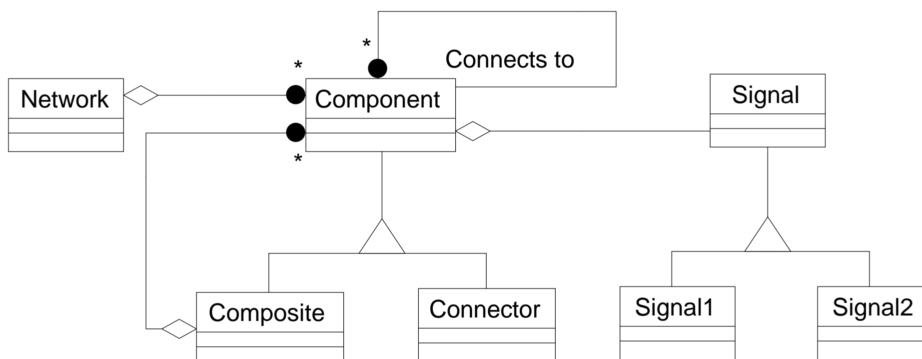


Figure 2, structure

When compared with Figure 1 the **`Adder`**, **`VolumeControl`**, **`Microphone`**, **`LFFilter`** are all representatives of the Connector class, **`LFChannel`** and **`HFChannel`** are representatives of the Composite class.

## Participants

- **Network**

  Is responsible for setting up the structure that is the network i.e. creating the components and making the connections.

  Acts as a high level interface environment including providing the input and querying the output.

- **Component**
  Declares the interface for objects in the composition
  Implements default behaviour for the interface common to all classes

- **Composite**
  Defines behaviour for components that consist of other components
  Abstracts and encapsulates a "sub-network"

- **Connector**
  Defines behaviour for primitive objects in the composition

- **Signal, Signal1, Signal2**
  Defines the information flow between Connectors
  Implements default behaviour for the information flow

## Collaborations

The Network class instance puts the input data to the appropriate starter Connectors and then asks the last Connector representative in the chain to calculate the output. The last Connector in the chain calls the connectors upon which it depend to forward their results for final computation. Each Connector when asked for output calculates its output and puts in the aggregate Signal class and returns a reference for this Signal class.

## Consequences

Calculations in the Network pattern can consume a lot of stack space in a large network.
Using a single data type in the Signal class to achieve flexibility can impose an overhead of data transformations.

## Implementations

1. Signals must share interface, but some variation adopting a signal class hierarchy (see Signal, Signal1, Signal2 in Figure 2) can be implemented. This implies that data can be different but interface as defined in Signal must be shared.
2. Connections from many connectors to one connector can be made using various collection classes. If the sequence by which the connections are made are important then use an ordered collection class like a stack or list. If the order is not important then use e.g. a set.
3. Another variation is changing the network dynamically. The network can change dynamically by changing the connections between all the components by adding appropriate interface for dynamically updating the connections e.g. Connect and Disconnect. And if the connections in each component are implemented using container classes as mentioned in *2* then it's a matter of deleting and adding connections.
4. Calculations in each component might be implemented using the strategy pattern. This could allow for just one class representing the Connector class which has to support some kind of generic calculation interface that can be implemented in a strategy pattern.
5. It might be desirable to differentiate input or output e.g. having a control signal that affects the other input signals. This can be obtained by enhancing the interface of the relevant component either by the method mentioned in 1 or by separate interface allowing references to the control signal.
6. Concurrency can be implemented to achieve better performance and possibly spread computation to different processors. Computation in each node (Component) can be calculated

in a separate thread and semaphors can be used to signal that computations have ended and that the nodes which depend on this node can obtain the result.

7. Instead of having calculations redone at every request, the network pattern can reset a flag in each Component instance which in turn are set when the first calculation have been done and data stored in the associated Signal class.
8. In C++ the errorhandling can be implemented using the exception handling mechanism and let the Network class catch the Network related errors.

## Sample Code

The following code shows an implementation of the very simple digital signal processor depicted in Figure 1. Note code has been replaced with an ellipsis "`. . .`" where the code is considered superfluous.

The LF and HF VolumeControls increase or decrease the signal and the two signals are finally added in the Adder. The LF-Channel and HF-Channel illustrate the use of the Composite class of the Network Pattern to model a sub-network.

The signal that passes through this network is a number of discrete frequencies with an associated value (volume i.e. sound pressure) comprised in a class Spectrum and contained within a Signal. The Signal class corresponds to the Signal class in the pattern description.

The Signal class is the sole representative for the Signal hierarchy in the Network pattern. Signal in this case wraps the Spectrum and provides access through member functions.

```
class Signal {
public:
      Signal( const Spectrum& v );
      ~Signal( );
      void SetValue( const Spectrum& v );
      Spectrum& GetValue();
      ostream& operator << ( ostream& s );
private:
      Spectrum Value;
};
```

The Component class is a true virtual base class defining the interface and default behavior of all elements in the network. The member function `SetInput()` is only used in subclasses that act as starters which in this case is the Microphone.
A simple mechanism has been implemented to avoid double (or more) recalculation of the same result in the case where a Component derived subclass provides it's output to more than one element. The mechanism is not applicable in concurrent environments. The private member flag `fReset` shows if computation have been done or if it has to be done. The overall network class, which in this case is the Chip (see subsequent class declarations), have the responsibility to reset using the `Reset()` member of all the components that are part of the network. When the member `GetOutput()` of an element is called and the calculation is done the first time, the flag is set using the member `Ready()`.
The other private member `Owner` is added to allow a component to add itself and possibly notify a network of problems

```
class Component {
      public:
            Component( Chip * o );
```

```
        virtual ~Component() {};
        virtual void Connect( Component * c ) = 0;
        virtual void DisConnect( Component * c ) = 0;
        void Reset();
        void Ready();
        virtual void SetInput( const Spectrum& val );
        virtual Signal& GetOutput() = 0;
    protected:
        BOOL GetIsReady();
    private:
        BOOL fReset;
        Chip *Owner;
    };
```

The Microphone is a starter component which mean that it does not depend on any other components hence the connector members have empty bodies. But other components depend on it.

```
    class Microphone : public Component {
    public:
        Microphone( Chip * o, const Spectrum& val );
         virtual ~Microphone();
        virtual void Connect( Component * c ){};
        virtual void DisConnect( Component * c ){};
        virtual Signal& GetOutput();
        virtual void SetInput( const Spectrum& val );
    private:
        Signal sign;
    };
```

The volume control component is a regular primitive in this network and calculations have for simplicity been implemented as a multiplying factor that is applied to all members of a spectrum meaning that all frequency values are amplified with the same value.

```
    class VolumeControl : public Component {
    public:
        VolumeControl(Chip * o, float val );
        virtual ~VolumeControl();
        virtual void Connect( Component * c );
        virtual void DisConnect( Component * c );
        virtual Signal& GetOutput();
        virtual void SetMultValue( const float val );
        virtual float GetMultValue();
    • • •
    };
```

The filter class has for simplicity been implemented with multiply factors for each frequency thereby giving various frequencies higher weight.

```
    class Filter : public Component {
    public:
        Filter( Chip * o, const float * val, const int count
    );
        virtual ~Filter();
        virtual void Connect( Component * c );
        virtual void DisConnect( Component * );
```

```
      virtual Signal& GetOutput();
      virtual void SetFilter( const float * val, int count
);
      virtual float * GetFilter();
• • •
};
```

The Channel class encapsulates a simple network consisting of a Filter and a VolumeControl. This illustrate the use of the Composite class to create sub-networks in the Network pattern. The Channel class can be reused since sub-network consisting of the Filter and The VolumeControl occurs twice in Figure 1.

```
class Channel: public Component {
public:
    Channel( Chip * o
           , const float * FiltVal
           , const int FiltCount
           , float VCVal );
      ~Channel();
      virtual void Connect( Component * c);
      virtual void DisConnect( Component * );
      virtual Signal& GetOutput();
• • •

private:
      VolumeControl* VolCon;
      Filter* Filt;
      • • •
};
```

The constructor of the Channel class creates the components that makes up the sub-network and interconnects the components.

```
Channel::Channel( Chip * o
               , const float * FiltVal
               , const int FiltCount
               , float VCVal )
      : Component( o )
{
  VolCon = new VolumeControl( o, VCVal );
  Filt = new Filter( o, FiltVal, FiltCount );
  VolCon->Connect( Filt );
}
```

The Adder class mimics the connection of different signals and adds the signals to the final output.

```
class Adder : public Component {
public:
      Adder( Chip * o  );
      virtual ~Adder(){};
      virtual void Connect( Component * c );
      virtual void DisConnect( Component * );
      virtual Signal& GetOutput();
      int Count();
• • •
```

```
};
```

The `Chip` class is the base class for the network class hierarchy and keeps track of all the elements using a template collection.

```
class Chip {
public:
      Chip(){};
      virtual ~Chip();
      virtual Signal& GetOutput( Signal *Input ) = 0;
      virtual void AddComponent( Component * c );
      virtual void DeleteAllComponents();
      virtual void SetStarter( Component * c );
      virtual void SetEnd( Component * c );
      virtual void ResetNetwork();
      virtual Component * GetStarter();
      virtual Component * GetEnd();
• • •
};
```

The `SampleChip` is the realization of a chip with the layout of elements as depicted in Figure 1. The class only overrides the `GetOutput()` member of the Chip class.

```
class SampleChip : public Chip {
public:
      SampleChip();
      virtual ~SampleChip();
      virtual Signal& GetOutput( Signal *Input );
      // set input in starters, get output from last
component
};
```

The network in Figure 1 is created in the constructor of `SampleChip` where all elements are created and interconnected. Note the use of the Channel class to create the sub-network consisting of the Filter and VolumeControl.

```
SampleChip::SampleChip()
: Chip()
{
Spectrum starter;
Microphone * mic;
VolumeControl *PreVC;
Channel * LF;
Channel * HF;
Adder * Sum;

      // initialize the starter
      • • •

      // create chip components,
      // note: they are automatically added as components
      mic = new Microphone( this, starter );

      // . . . are arguments being characteristics of
      // VolumeControl, LF and HF
```

```
        PreVC = new VolumeControl( this, . . . );
        LF = new Channel( this, . . . );
        HF = new Channel( this, . . .  );
        Sum = new Adder( this );

        // create the connections
        SetStarter( mic );    // Network initialized
        PreVC->Connect( mic );
        LF->Connect( PreVC ); // split
        HF->Connect( PreVC );
        Sum->Connect( LF );  // join
        Sum->Connect( HF );
        SetEnd( Sum );        // Network ends
    }
```

An example of retriving the output is:

```
    SimpleChip *aChip;
    Spectrum    starter;
    Signal*     InputSignal;
    Signal&     ResultSignal;

        aChip = new SimpleChip();
        // initialize starter
        InputSignal = new Signal( starter );
         ResultSignal = aChip->GetOutput( s );
```

## Known Uses

AVS, a component-based software environment for visualizing complex data and for building applications with interactive visualization and graphics functions uses a variant of this pattern in organizing modules (file readers, data processing etc.) into a network that can be manipulated by a visual network editor. AVS employs an object-oriented visual programming interface to create, modify and connect these objects. The Visual Network Editor allows developers to construct applications as connected, hierarchical networks of objects, and to create and modify data structures with "drag-and-drop ease".

Robert Engel, Hermann Hüni and Ralph Johnson describes in [Eng95] a framework "*Conduits+*" for network protocol software where software components can be reused across layers (eg. TCP and IP). The paper uses the term Conduits for a software component that can be connected on two sides. Conduits are bidirectional meaning that both sides can communicate to it's neighbors. The design of the framework were narrowed down to use only four kinds of Conduits which all have one neighbor on side A, and the Mux Conduits can have many neighbors on side B, an Adapter has no neighbor conduit on side B and ConduitFactory and Protocol have exactly one neighbor conduit on side B. The framework has been used to implement multiple layers of a signalling system within a flexible multi-protocol ATM access switch being marketed today.

Oticon currently consider using this pattern to simulate Digital Signal Processors for hearing aids in their fitting software Otiset. It was a requirement that the components of the simulation that they were reuseable and able to handle multiple input and output signals.

## Related Patterns

The composite pattern in the [Gam95] book by the Gang of Four are related, it does however implies a tree structure and doesn't take into account input/output relations. But it does somewhat resemble the hierarchy of the parts.

There is a reference in [Gam95] book that states that a variant of the Mediator pattern CSolver does something similar to the Network pattern. The network pattern is different because the flow has been distributed to the objects and not centralized to the mediator object.

The Facade pattern are also related in the sense that *the Network class* act as a high-level interface.

The Strategy pattern can be used as mentioned under *implementations 4* for calculations in the connector class thus decoupling the calculation from the networks Component elements.

Mary Shaw have in [Vlis96] in her paper "*Some Patterns for Software"* a one page informal description of a very similar pattern: Pipeline.

Regine Meunier describes *The Pipes and Filters* architectural pattern in [Busch96]. This is a very general approach to the subject and covers most aspects of these sorts of patterns. The main component in the *Pipes and Filters* paper are Filters that enrich, refine or transform input data, Pipes that serve as connectors between Filters, Data Sources that provide the system with input, and Data Sink such as a file, terminal, etc. that consumes the output. The activity of a filter can be passive or active. If the filter is passive it can gain it's data either by a subsequent pipeline element pulls output data from the filter or that a previous pipeline element pushes new input data to the filter (push vs. pull). If the filter is active it exist on it's own with a loop pulling it's input from and pushing its output down pipelines. So an active filter is typically a program or a thread. In this context the Network patterns employs a passive filter (or pull) mechanism for activating, because each component (comparable with filters) in the the Network pattern ask it's components of which it depends for their output and so on.

**References:**

Gam95: Gamma, Helm, Johnson and Vlissides "*Design Patterns",* Addison-Wesley 1995

Busch96: Buschmann, Meunier, Rohnert, Sommerlad and Stal *"Pattern Oriented Software Architecture: A System of Patterns"*, John Wiley & Sons, 1996

Vlis96: Vlissides, Coplien and Kerth "*Pattern Languages of Program Design 2"*, Addison-Wesley 1996

Eng95: Engel, Johnson and Hüni, "*A Framework for Network Protocol Software"*, OOPSLA95