## Abstract

Chaotic dynamical systems characteristically exhibit erratic, seemingly random, behavior. They are highly sensitive to initial conditions. Any alteration to the state of a chaotic system will eventually lead to very large differences in behavior. Until recently, the preferred approach to chaotic dynamical systems has been to avoid them. A method developed by Ott, Grebogi, and Yorke (OGY) takes advantage of this sensitivity to initial conditions, and other characteristics of chaotic systems, to produce periodic behavior from chaotic behavior. The pattern language provided in this paper describes the component steps to implementing the OGY method for controlling chaotic dynamical systems. It establishes an architecture for chaos control software that allows for experimentation with algorithms and data structures at each step in order to improve the performance of the overall system.

# A Pattern Language for Controlling Chaotic Activity

Gregg T. Stubbendieck

July 21, 2002

## 1 Introduction

The study of chaotic dynamical systems spans a range of disciplines including atmospheric sciences, physics, engineering, mathematics, and neurosciences. The behavior of a chaotic system is by nature unpredictable over long periods of time since the smallest change to the system may result over time to large changes in behavior. This is the famous "butterfly effect," in which is speculated that the flap of a butterfly's wings might over time change the course of global weather patterns. There are many problems where it is desirable to control the underlying chaotic dynamics of a system.

After years of study, both theoretical and experimental, much is known about chaotic dynamical systems. In particular, it is known that even though a chaotic system behaves in a seemingly random way there is a structure to the underlying behavior, and that structure has certain characteristics. Edwared Ott, Celso Grebogi, and James A. Yorke (OGY) developed a method for controlling a chaotic dynamical system[6][1] through small manipulations of the system. The method exploits the structure of the underlying dynamics and the sensitivity of a chaotic system to small changes to produce predictable periodic behavior.

The essence of the OGY method for chaos control is fairly straightforward. Once the control parameters have been calculated it is simply necessary to wait for the system to reach a region of state space that is covered by the control parameters, then to issue small modifications to a control parameter to ensure that the system will come back to the same controllable region in a predictable amount of time.

A software engineer is interested in finding reusable code whereever possible. Since the OGY method exploits characteristics common to all chaotic systems it should be possible to write the controller component once and to use it for all chaotic systems. In order to accomplish this goal it is necessary to answer two challenges. First, the unique characteristics of a particular chaotic system may require special processing before reaching the OGY specific components. Any special reshaping or filtering of data should occur outside the implementation of the core control component. Second, it is necessary to have flexibility in the method of computing the OGY control parameters. Assuming that an analytical solution for the chaotic system in question is not available, which would be the common case for real world chaotic systems, the control software must calculate the required information from data sampled from the system. It is not clear what method for calculating this information is best. The notion of "best" implies a set of ranking criteria that may change depending upon the goals of a particular controller situation. The overall controller system should be organized so that control parameter computation methods may be changed with minimum impact to the rest of the system.

## 1.1 The Purpose of the Patterns

The pattern language provided in this paper describes the component steps to implementing the OGY method for controlling chaotic dynamical systems, continuing the work in [7]. It establishes an architecture for chaos control software that allows for experimentation with algorithms and data structures at each step in order to improve the performance of the overall system. Rather than specifying what is done in a particular experimental instance with custom built software, this pattern language separates problem specific considerations from the more general aspects of OGY control. By establishing a protocol for communication between components, the architecture allows flexibility in the implementation while improving reusability. This independence may be exploited in an asynchronous parallel execution environment, for example, by transparently allowing the more computation intensive aspects of the OGY method to execute on separate processors while time critical functions, such as sampling, continue to operate simultaneously.

## 1.2 What is Controlling Chaos?

The basic objective of control is to manipulate a system in a way that produces a predictable result. To change the behavior of a stable, non-chaotic, system, it is necessary to manipulate the system in a way that is proportional to the amount of change desired. For example, in order to lift a rocket off the ground, the rocket motor must produce sufficient thrust to offset the weight of the rocket. The amount of thrust required varies directly with the weight of the rocket.

Chaotic systems are not stable. They are highly sensitive to even the slightest changes in state. Any modification to a chaotic system, no matter how small, results in large changes in behavior over time. Lorenz[5] observed this sensitivity in numerical experiments in weather prediction using the system of equations given by (1)–(3).

$$\frac{dx}{dt} = -\sigma x + \sigma y \tag{1}$$

$$\frac{dy}{dt} = -xz + rx - y \tag{2}$$

$$\frac{dz}{dt} = xy - bz \tag{3}$$

The state trajectory of the Lorenz equations, produced by numerically integrating the previous system of equations, is shown in Figure 1. As illustrated in Figure 2, small differences in the initial conditions of numerical experiments lead to radically different behavior over time. In contrast to a stable system, where a change in initial conditions leads to a proportional change in behavior, any change in the initial conditions of a chaotic system lead to long term behavior that cannot be predicted.

A chaotic system is considered to be controlled if it is made to exhibit quasi-periodic[5] behavior. Quasi-periodic behavior may be loosely visualized as the trajectory of a race car on a circuit course, where the system of interest is the race car and its state is represented as the position of its center of mass on the track. A truly periodic trajectory might be traced down the middle of

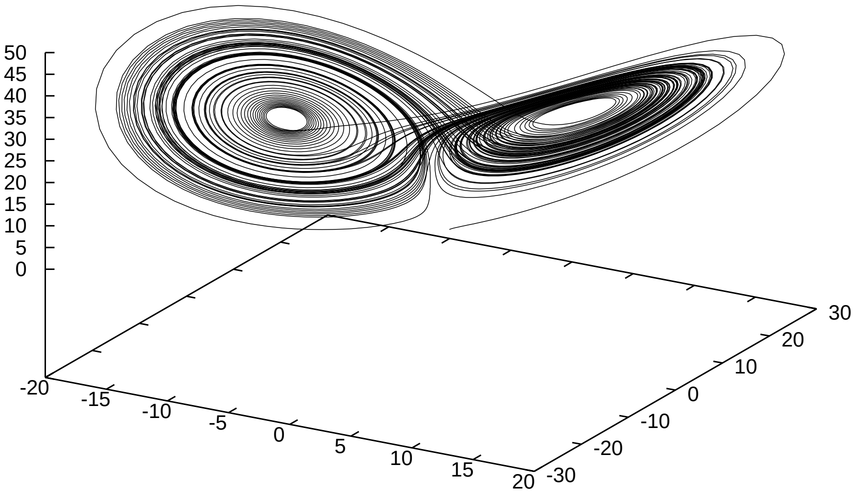

Figure 1: The state space trajectory Lorenz system of equations shows the evolution of the system over time. The point plotted at time $t$ is $(x_t, y_t, z_t)$. The chaotic nature of the system of equations means that the trajectory will never repeat itself as it traverses this shape.

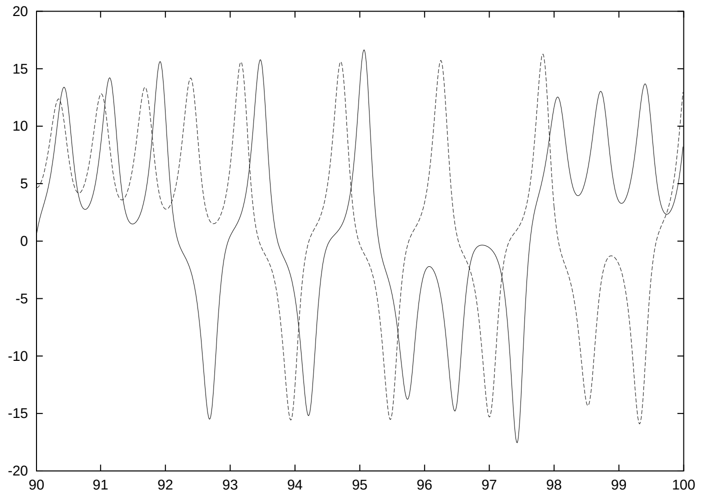

Figure 2: Graph of the $x$ coordinate of the Lorenz equations over time with $\sigma = 10$, $r = 28.5$, and $b = 8/3$. The solid line is produced with an initial condition of $(1, 1, 1)$, and the dashed line is has an initial condition of $(1.00000000001, 1, 1)$. The horizontal axis is time. The plot begins 90 time units after the beginning of the experiment.

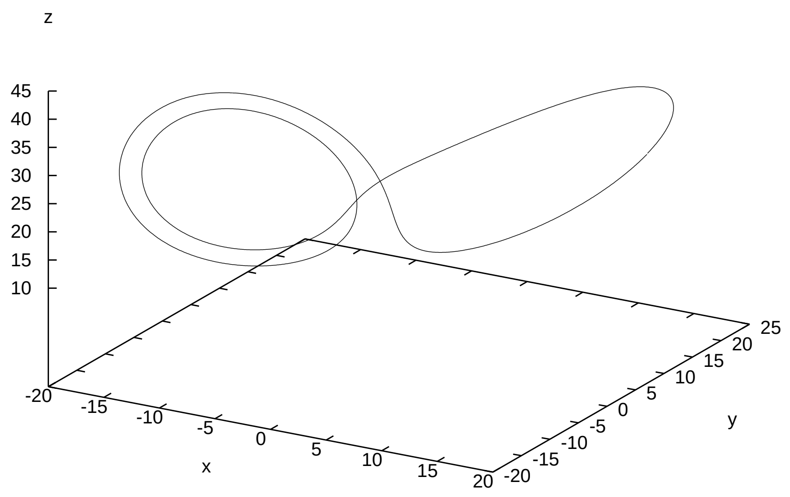

Figure 3: Nearly periodic behavior in the Lorenz system. The system appears to repeat its trajectory but is actually just passing very near to itself. If left alone, the Lorenz system will eventually leave the region of the periodic trajectory and revert to unpredictable behavior. If the system is held close to this trajectory its future state will be predictable, and the system will be considered to be controlled.

the course, but the race car wanders freely from side to side on the track as it traverses the circuit. Since the race car stays within a bounded distance of the true circuit its behavior is said to be quasi-periodic. The the OGY method is motivated from the observation that a chaotic system occasionally exhibits behavior that looks quasi-periodic before reverting to more unpredictable behavior. Figure 3 illustrates such behavior in the Lorenz attractor. Such nearly periodic behavior suggests the location of a truly periodic trajectory nearby in the state space since two trajectories passing very near to each other will be nearly parellel for a period of time.

## 1.3 OGY Control

To this point, the discussion has focused on continous systems where the state of the system evolves smoothly over time. If the state of the system is plotted over time, its track does not contain any breaks. However, the OGY method is formulated in terms of a a discrete map. Rather than producing a continuous trajectory, a discrete map shows the point position of the system at each iteration of the map.

As will be illustrated in the Apply a Poincaré Map pattern, a continuous chaotic system may be analyzed as a discrete chaotic map. Roughly speaking, a surface, called a surface of section, is placed in the region of the chaotic activity. The surface of section is positioned so that all chaotic trajectories will pierce through it. The Poincaré map produces a sequence of points

$x_1, x_2, \ldots, x_n, \ldots$. The point $x_n$ is produced at the $n$th iteration of the map when the trajectory of the underlying continuous system pierces through the surface for the $n$th time. The coordinates of $x_n$ are relative to the surface. If the underlying continuous system is in $d$-dimensional space, the points on the Poincaré map will be in the $(d-1)$-dimensional space of the surface of section. The position of the point on the surface piercing when a trajectory passes through the surface.

The behavior produced on a Poincaré map can be used to interpret behavior in the continuous system. For example, if

$$|x_{n+1} - x_n| < \epsilon,$$

where $\epsilon$ is very small, then it may be said that the continuous system is exhibiting quasi-periodic behavior–at least for time period covered by the $(n+1)$th iteration of the map.

A controller using the OGY method must identify such points as estimates of actual fixed points. Once fixed point estimates have been identified, the behavior of the attractor near the fixed points must be sampled. For each point $x_k$ that falls near a fixed point estimate, the pair $(x_k, x_{k+1})$ is recorded. Since the behavior of the system changes smoothly with respect to position in state space, the behavior of the system in a small region around a fixed point may be linearized to provide a useful estimate of system behavior. The sampled pairs of points are used to calculate such a linearization in order to determine which regions of state space near the fixed point tend to move toward it and which regions tend to move away. The small neighborhood over which the linearization applies is the controllable region of the fixed point.

Having located a fixed point and linearized the system behavior around it, it is necessary to provide an input to the chaotic system that perturbs its behavior slightly. In the Lorenz system of equations, the $r$ value of (2) may be modified slightly. The effect of a small perturbation should be to slightly warp the attractor, changing the location of the fixed point in state space.[1] The fixed points of the perturbed system are estimated from sampled data and are matched up with fixed points of the unperturbed system. The movement of the fixed point with respect to the perturbation is estimated from these paired points using linear interpolation.

At this point, the OGY control method, as illustrated in Figure 4 may be engaged. The chaotic system is sampled for a period of time until it falls into the controllable region of a fixed point. If the point lands in an area that tends to move away from the fixed point the control parameter is manipulated so that the system tends to move toward the fixed point in the in the perturbed system, keeping the system in the controllable region.

## 2 Patterns

### 2.1 Control a Chaotic System with OGY

#### 2.1.1 Context

You are faced with a chaotic dynamical system which you desire to behave in a predictable way. A control parameter of the system is avialable to modify the behavior of the system in some way.

[1]More exotic changes to the attractor are possible for small changes to the control parameter, and it is not clear at the outset what range of parameter changes is appropriate for a particular chaotic system. Some experimentation is required to find the appropriate parameter change size.

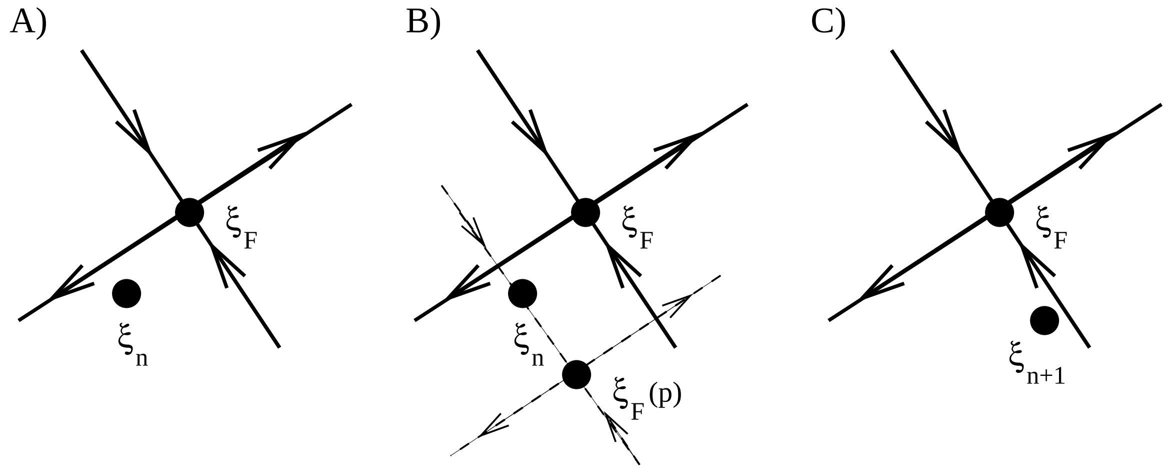

Figure 4: Conceptual operation of the OGY control method. Arrowheads indicate the direction of movement of successive iterates on the discrete map. (A) A point falls near a fixed point of the discrete chaotic system. (B) The control parameter is perturbed so that the point is in the stable region of the perturbed fixed point. (C) The next iteration of the system again falls near the original fixed point.

Sensors on the chaotic system periodically produce a vector samples of the system state in numeric form.

### 2.1.2 Problem

What is the basic structure of the relationship between a controller using the OGY method and the chaotic dynamical system?

### 2.1.3 Forces

- To avoid code tailored to a specific chaotic system within the controller component, the interface between the control component and the chaotic system must be expressed in the most general terms acceptable to the controller.
- A time series of samples of the chaotic system state and a control parameter together are sufficient to compute control parameters and then establish control.

### 2.1.4 Solution

Figure 5 shows the high level design of the OGY system. The `ChaoticSystem` class provides the software interface that is expected by the `OgyController`. Samples are obtained via the `InstrumentListener` interface. The underlying chaotic system is manipulated by the controller through the `setPerturbation(Real p)` method. The `Real` parameter represents the real valued amount by which the `ChaoticSystem` is to be perturbed by the `OgyController`. The `InstrumentListener` interface provides the mechanism for the `ChaoticSystem` to communicate the samples read by its instrumentation devices to interested parties. Samples are communicated

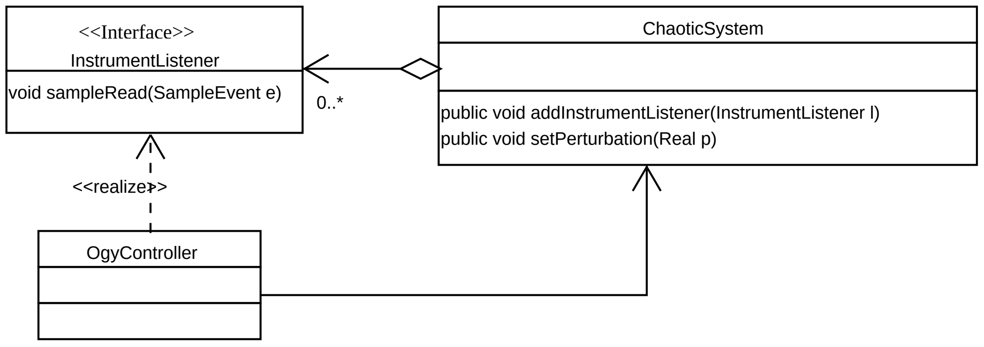
<<Interface>>
InstrumentListener
void sampleRead(SampleEvent e)
0..*
ChaoticSystem
public void addInstrumentListener(InstrumentListener l)
public void setPerturbation(Real p)
<<realize>>
OgyController

Figure 5: A chaotic dynamical system and its OGY controller

from the `ChaoticSystem` to its listeners in the raw form retrieved by the instruments. In Figure 5, the `OgyController` receives samples directly from the `ChaoticSystem` via the `Instrument-Listener` interface.

This pattern serves to illustrate the conceptual structure of the OGY method. The simplified view of the OGY control process provided in Figure 5 implicitly assigns responsibilities to the `OgyController` that are not directly related to the control function. The raw sequence of samples received from a chaotic system must often be transformed before it is useful to an OGY controller. For example, if the dynamics of the chaotic system are continuous, then the samples will represents points on a continuous trajaectory. It would be necessary for the controller to apply a Poincaré map to the data stream in order to produce the discrete map that is assumed by the OGY control mechanism as discussed in Section 1.3. The following patterns serve to separate such implied responsibilities out of the `OgyController` class to provide a flexible framework for applying the OGY control method.

### 2.1.5 Related Patterns

The `InstrumentListener` is an Observer[2] that is realized in the `OgyController` class. The basic responsibility of the `ChaoticSystem` class is as an Adapter[2] pattern, converting the interface expected by the `OgyController` to that of the underlying chaotic system. Depending upon the implementation of the underlying system, the `ChaoticSystem` class may be a component in a larger wrapper-like class such as Bridge, Decorator, Facade, or Proxy, also of [2].

The `SampleEvent` class is a subclass of `java.util.EventObject`. A `SampleEvent` is passed to the `sampleRead()` method of an `InstrumentListener` when a sample is received from the `ChaoticSystem`. A `SampleEvent` object contains a data item in the form of a `Point` object representing the vector of salar samples received.

### 2.1.6 Example Code

```
public class OgyController implements InstrumentListener {
   private ChaoticSystem chaoticSystem;

   private Real computeNextPerturbation(Point e)
   {/* OGY control implementation goes here */}
   public void sampleRead(SampleEvent e) {
      Real perturb = computeNextPerturbation(e.getPoint());
      chaoticSystem.setPerturbation(perturb);
}  }
```

## 2.2 Time Series Data Filter

### 2.2.1 Context

In order for a computer program to perform feedback control such as is performed in the OGY method, it is necessary to obtain a representation of the system activity from periodic measurements of system state as illustrated in Control a Chaotic System with OGY. It is not always the case that the vector of samples received via the `InstrumentListener` interface is in a form that is directly usable for the OGY method in its most general form.

### 2.2.2 Problem

Requiring the `OgyController` class to perform necessary data transformations weakens it by diluting its focus on the specific task of carrying out OGY control and by tightening the coupling between the chaotic system and the controller. If the `OgyController` is tightly coupled with the `ChaoticSystem` its implementation will tend to be specialized to the ChaoticSystem of interest.

### 2.2.3 Forces

- Hard coding the data filtering role into the `OgyController` reduces its flexibility.
- Indirect interfaces may be slower than direct interfaces. Response time is critical in real-time systems, leading to a tendency to write specialized code.
- It may be necessary to apply a number of transformations to a raw data stream to convert it from the form produced by a `ChaoticSystem` to one usable by `OgyController`.

### 2.2.4 Solution

Reduce the coupling between the `OgyController` and the `ChaoticSystem` by moving data filtering responsibilities out of the `OgyController`. Such a modification allows it to focus more clearly on its specific task of carrying out OGY control. To loosen the coupling, introduce a chain of one or more `Filter` objects between the `ChaoticSystem` and the `OgyController`.

Figure 6 depicts the architecture as modified to allow data filters between the `ChaoticSystem` and the `OgyController`. The `InstrumentListener` interface serves the same purpose as in Figure 5. The `OgyController` has been modified to be a `TimeSeriesListener` so that it is no longer able to register directly as a listener to the `ChaoticSystem`. A distinction is made in the depicted design between data received from an external source as provided through the `InstrumentListener` interface, which has a representation determined by the external source, and data provided through the `TimeSeriesListener` interface with a format that is determined by the control application. The data received by `OgyController` must be in the expected format to avoid artificial code dependencies in its implementation.

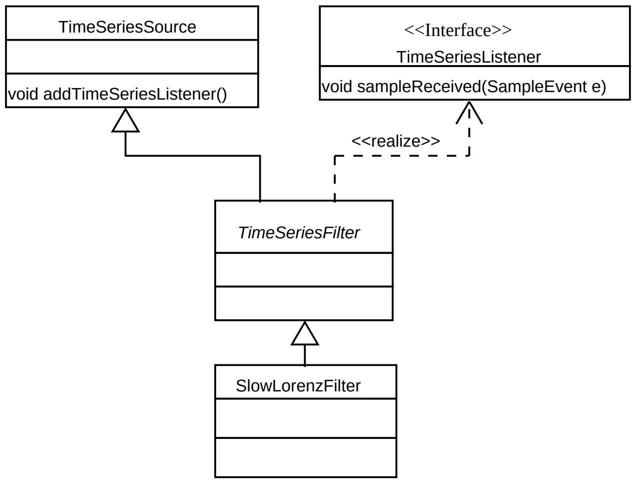

Figure 6: The `SlowLorenzFilter` is a `TimeSeriesFilter`. A `TimeSeriesFilter` is both a `TimeSeriesListener` and a `TimeSeriesSource`.

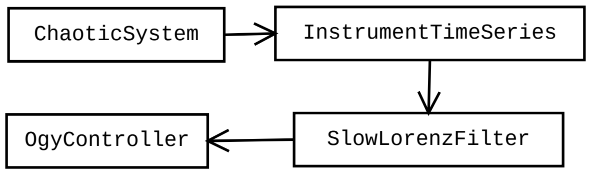

Figure 7: A block diagram showing a possible sequence of `TimeSeriesFilter` objects between the `ChaoticSystem` and the `OgyController`.

The `TimeSeriesSource` class provides the interface for `TimeSeriesListener` objects to subscribe as listeners with objects providing time series data. The `InstrumentTimeSeries` class is both an `InstrumentListener` and a `TimeSeriesSource`. An object of a class derived from `InstrumentTimeSeries` translates the external data format of the `ChaoticSystem` to the internal format expected by the controller application. The `TimeSeriesFilter` class is an abstract class derived from `TimeSeriesSource` and `TimeSeriesListener`.

A `TimeSeriesFilter` is both a `TimeSeriesListener` and a `TimeSeriesSource`. A `TimeSeriesFilter` receives a data stream from another `TimeSeriesSource`, modifies the data, then sends the modified data to its own `TimeSeriesListeners`. The `SlowLorenzFilter`, for example, passes on only every other sample that it receives. `TimeSeriesFilters` are used to change the data stream produced by the `ChaoticSystem` to a form usable by the `OgyController` without requiring modification to either the `ChaoticSystem` or `OgyController` software component.

The block diagram in Figure 7 shows an example flow of information through a series of objects based on the classes depicted in Figure 6.

### 2.2.5 Related Patterns

The `TimeSeriesFilter` combines the Observer patterns with a hint of the Adapter, both from [2]. Rather than converting class interfaces, as done by the Adapter, it converts a data stream from one form to another. The class is a realization of the Observer pattern. It registers with `TimeSeriesSource` to receive data. Since it is derived from `TimeSeriesSource`, it is also a Subject, allowing other Observer objects to regeister with it and receive its transformed data stream.

The `InstrumentTimeSeries` class has similar pattern relationships. It actually implements the Adapter pattern, translating the interface of `InstrumentListener` to that of `TimeSeriesListener`.

### 2.2.6 Example Code

The `LorenzSystem` class represents a `ChaoticSystem`.

```
public class LorenzSystem extends ChaoticSystem {

   // ... (some code not shown)

   /**
    * Integrate one timestep into the future and notify
    * listeners of the new sample.
    */
   public void integrateNext() {

      // Not shown: compute the next point in the trajectory,
      // yNext, using a numerical integration technique.

      incrementTime( timestep );
      Sample s = new Sample(yNext, getCurrentTime());
      SampleEvent e = new SampleEvent(this, s);
      notifyNewSample(e);
} }
```

The `LorenzTimeSeries` class handles the translation from the `LorenzSystem` output format and the application internal time series data format. Since the `InstrumentListener` and `TimeSeriesListener` data formats for this example are equivalent, the class merely serves as an Adapter between the `InstrumentListener` and `TimeSeriesListener` interfaces.

```
public class LorenzTimeSeries extends InstrumentTimeSeries {
   public LorenzTimeSeries() {}

   public void readSample(SampleEvent e) {
      /* in this example, the instrument data format is
       * the same as the time series data format
       */
      notifyNewSample(new SampleEvent(this,sampleOut));
}  }
```

The `TimeSeriesFilter` class provides basic implementation for objects filtering time series data. It declares an abstract method `filterSample(Sample s)`, which a subclass will use to implement its particular sampling strategy

```
/**
 * A TimeSeriesSource which receives samples from another
 * TimeSeriesSource, performs some transformation on the
 * received samples, then passes the transformed samples
 * on.
 */
public abstract class TimeSeriesFilter extends TimeSeriesSource
   implements TimeSeriesListener {
   public TimeSeriesFilter() {}
   /**
    * A sample has been read from the instrumented device.
    * Transform the data to the TimeSeries format and send the
    * sample along to TimeSeriesListeners.
    * @param e the object associated with this event, containing
    *        the sample value.
    */
   public void sampleReceived(SampleEvent e) {
      Sample sampleIn = e.getSample();
      filterSample(sampleIn);
   }
   /**
    * Receive the next sample in the time series.  When
    * transformed samples are produced by the filter, distribute
    * them to this object’s listeners.
    * @see TimeSeriesSource#notifyNewSample()
    */
   protected abstract void filterSample(Sample sample);
}
```

The `SlowLorenzFilter` illustrates time series filtering by discarding every other sample it receives.

```
public class SlowLorenzFilter implements TimeSeriesFilter {
   private int count = 0;

   protected void filterSample(Sample s) {
      if ( count % 2 == 0 ) {
         notifyNewSample(new SampleEvent(this,s));
      }
      ++count;
}  }
```

The `LorenzEx` class initializes a `LorenzSystem` object then creates a `LorenzTimeSeries` object and registers it as an `InstrumentListener` with the `LorenzSystem`. A `SlowLorenzFilter` object is created and registered as a listener to the `LorenzTimeSeries` object. Finally, an object of the inner class `LorenzListener` is registered with the `SlowLorenzFilter` object to print out the filtered time series that it receives.

```
public class LorenzEx {
   private class LorenzListener implements TimeSeriesListener {
      public LorenzListener() {}
      public void readSample(SampleEvent e) {
         Sample s = e.getSample();
         double[] pt = s.getPoint();
         System.out.println(""+pt[0]+" "+pt[1]+" "+pt[2]);
   }  }

   private void go() {
      double[] yInit = new double[3];
      for (int i=0; i<3; i++)
         yInit[i] = 1.0000000000;
      LorenzSystem ls = new LorenzSystem(yInit,0.01);
      TimeSeriesSource lts = new LorenzTimeSeries();
      ls.addInstrumentListener( lts );
      TimeSeriesSource slf = new SlowLorenzFilter();
      lts.addTimeSeriesListener( slf );
      slf.addTimeSeriesListener(new LorenzListener());
      for (int i=0; i<10000; i++)
      {
         ls.integrateNext();
}  }  }
```

## 2.3 Embed the Time Series

### 2.3.1 Also Known As

Delay Coordinates

### 2.3.2 Context

In sampling a nonlinear $n$-dimensional dynamical system, it is not always possible to sample all $n$ dimensions. In order to characterize the behavior of the system it is necessary to represent the system behavior in the appropriate number of dimensions.

### 2.3.3 Problem

How does one transform a scalar time series into a $n$-dimensional series of samples?

### 2.3.4 Forces

- Time delays between coordinates of the embedding should be multiples of the sample rate.
- It is not simple to calculate the optimal time delay or embedding dimension solely from sampled data.
- Downstream components should not care if they're getting samples from an embedded system or from the $n$-dimensional nonlinear system itself.
- It is undesirable to clutter the sampler with two jobs–obtaining samples from the dynamical system and performing the embedding.

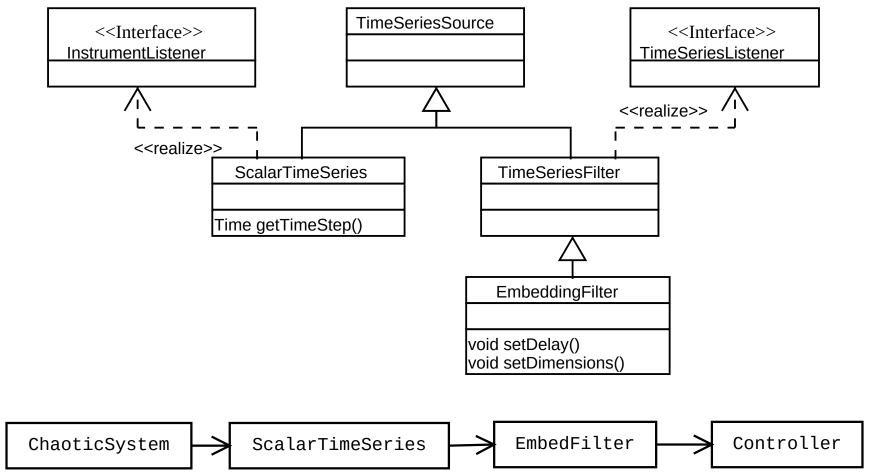

Figure 8: Embedding filter. Top: UML diagram depicting class relationships; bottom: block diagram of flow between components.

### 2.3.5 Solution

A theory due to Takens[8] states that the behavior of a chaotic system can be reconstructed from a scalar time series with delay coordinates. A $m$-dimensional point $X(t)$ with delay $T$ can be constructed from a scalar time series $z(1), z(2), \dots, z(t)$ using delay coordinates as follows.

$$X(t) = [z(t), z(t-T), \dots, z(t-(m-1)T)]^T \tag{4}$$

Figure 9 shows the Lorenz attractor reconstructed using delay coordinates from the $x$ dimension of the system. While the shape of the reconstructed system is not exactly the shape of the original, the Lorenz attractor can be recognized.

Figure 8 shows an `EmbeddingFilter` class derived from `TimeSeriesFilter`. It can be used to embed a 1-dimensional time series as part of a filter chain. Two operations for tuning the behavior of `EmbeddingFilter` are available. The `setDelay(int)` operation sets the delay between dimensions in the embedding. The `setDimensions(int)` operation sets the number of dimensions in the embedded data stream.

Calculation of the delay and dimensions parameters from sampled data is both computationally intensive and data intensive. On the one hand, it is desirable to embed the system in the smallest number of dimensions that faithfully represents the shape of the attractor. This reduces the complexity of computations later in the process. One method for finding the number of dimensions for which an embedded trajectory does not intersect with itself. The unavoidable presence of noise in the data stream adds a level of uncertainty to this calculation since noise may make it falsely appear that a trajectory crosses itself. On the other hand, the varying time delay between coordinates warps the shape of the reconstructed attractor. A good time delay value causes maximal separation between embedded trajectories.

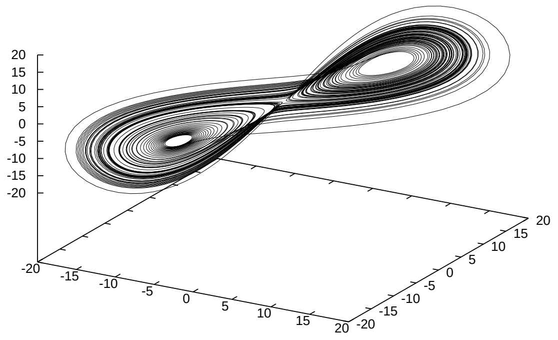

Figure 9: The Lorenz attractor reconstruced with delay coordinates using the $x$ dimension of the equation. The time delay between coordinates in this diagram is $0.06$.

### 2.3.6 Related Patterns

The `EmbeddingFilter` may be implemented as a specialization of `TimeSeriesFilter` from Section 2.2. Rather than addressing the general concept of filtering, this pattern is concerned with the conversion of a 1-dimensional time series into a $n$-dimensional time series that reconstructs the dynamical behavior of the underlying system.

### 2.3.7 Example Code

A `ScalarTimeSeries` object is used as the `InstrumentListener` applied to the `LorenzSystem` of the previous example. The `ScalarTimeSeries` class passes along only the first coordinate of the sample vector received from the `ChaoticSystem`, simulating the situation where only one sample dimension is available.

```
/**
 * A source of 1-dimensional time series data received from
 * an instrumented system.  In this implementation, an
 * instrumentation sample is received and only the first
 * dimsion of the received sample vector is passed along.
 */
public class ScalarTimeSeries extends TimeSeriesSource
   implements InstrumentListener
{
   /**
    * A sample has been read from the instrumented device.
    * Transform the data to the TimeSeries format and send the
    * sample along to TimeSeriesListeners.
    * @param e the object associated with this event, containing
    *        the sample value.
    */
   public void readSample(SampleEvent e) {
      Sample sampleIn = e.getSample();
      double[] pointIn = sampleIn.getPoint();
      double[] pointOut = new double[1];
      pointOut[0] = pointIn[0];
      Sample sampleOut =
         new Sample(pointOut,sampleIn.getTime());
      notifyNewSample(new SampleEvent(this,sampleOut));
}  }
```

The `EmbeddingFilter` class receives a time series of scalar samples and reconstructs $n$-dimensional dynamics using the method of delays. Once the number of dimensions and the time delay between coordinates have been decided, the job of computing the embedding is straightforward. Here, the embedding parameters are set from precalculated values. An adaptable `EmbeddingFilter` might calculate the parameters from sampled data.

```
public class EmbeddingFilter extends TimeSeriesFilter {

   // ....  (some code not shown)

   /**
    * Receive the next sample in the scalar time series.  When
    * transformed samples are produced by the filter, distribute
    * them to this object's listeners.
    *
    * @see TimeSeriesSource#notifyNewSample()
    */
   protected void filterSample(Sample inSample) {
      double newPoint = inSample.getPoint()[0];
      if (bufSize == buffer.length) {
         //
         // Circular buffer is full, generate an embedded point
         double[] outPoint = new double[numDims];
         for (int i=0; i<numDims-1; i++) {
            int sourceIndex = startIndex + i * delay;
            sourceIndex %= buffer.length;
            outPoint[i] = buffer[sourceIndex];
         }
         outPoint[numDims-1] = newPoint;
         /*
          * Create an output Sample object.  Use the latest
          * scalar sample's time as the embedded sample's time.
          */
         Sample outSample =
            new Sample(outPoint,inSample.getTime());
         notifyNewSample(new SampleEvent(this,outSample));
         // Bookkeeping:
         // replace oldest point in buffer with newest
         buffer[startIndex] = newPoint;
         // then move start index
         startIndex ++;
         // remember: we know bufSize == buffer.length here
         if (startIndex == bufSize) {
            startIndex = 0;
      }  }
      else // bufSize < buffer.length; fill the buffer
      {
         buffer[bufSize] = newPoint;
         ++bufSize;
}  }  }
```

## 2.4 Apply a Poincaré Map

### 2.4.1 Also Known As

- Surface of Section

### 2.4.2 Context

The OGY method operates on a discrete chaotic map. Periodic points are identified through observation of system behavior on the map. The dynamical behavior of the system near the periodic points is determined experimentally, both with and without perturbation of the control parameter. After sampling the local behavior of the periodic points, the behavior is linearized, and the parameters for control are calculated.

Many chaotic systems are continuous in nature. Periodic behavior in such systems is exhibited as traversal of periodic orbits in the system's state space. In order to gain control of the system with the OGY method, the continuous behavior must be converted into a discrete map.

### 2.4.3 Problem

How does one apply the OGY method to a continuous $m$-dimensional dynamical system?

### 2.4.4 Forces

- It is not simple to find an optimal placement for a Poincaré surface.
- The time delay between piercings of the Poincaré surface is not predictable in general. The notion of time must be modified or discarded.
- Finding periodic points on a discrete map requires less data and computation than finding periodic orbits in continuous data.
- It is possible to determine characteristics of a continuous system by analyzing the discrete behavior exhibited on a Poincaré map.

### 2.4.5 Solution

Apply a Poincaré map to the system to obtain a discrete time view of the continuous system's dynamical behavior. A surface of section is a $(m - 1)$-dimensional surface that is situated so that it is transverse to the flow on the attractor. The surface may be considered to be polarized, so that it has a back and a front. As depicted in Figure 10, each time the flow of the system pierces the surface from back to front, a point is plotted on the surface at the location of the piercing[4]. The resulting discrete map is a Poincaré map[3]. Periodic points on the surface of section correspond

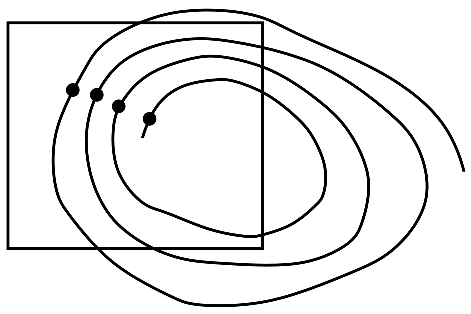

Figure 10: A Poincaré map is calculated by detecting when the flow of the continuous system pierces the surface of section.

to periodic orbits in the continuous system. Figure 11 depicts a closeup view of a Poincaré map of the Lorenz attractor.

The `PoincareFilter` class in Figure 12 implements a Poincaré map. The `ContinuousTimeSeries` receives $m$-dimensional samples from a continuous dynamical system and transmits them to the PoincareFilter, which translates the continuous time series into a $(m-1)$-dimensional discrete system suitable for OGY control.

### 2.4.6 Related Patterns

The `PoincareFilter` may be implemented as a specialization of `TimeSeriesFilter` from Section 2.2. Since the OGY method is defined for discrete chaotic systems, the ability to convert a continuous flow into a discrete map while preserving the required dynamical information is fundamental to the wide applicability of the OGY method.

### 2.4.7 Example Code

```
/**
 * Transforms a n-dimensional continuous time series into
 * a discrete time series of points on the surface of section.
 */
public class LorenzPoincareFilter extends TimeSeriesFilter {
   private boolean lastPointNegative = false;
   private Sample lastSample = null;

   protected void filterSample(Sample inSample) {
      if ( surfacePierceDetected() ) // just pierced through {
         double[] pt = inSample.getPoint();
         double[] newPt = new double[inSample.getNumDimensions()-1];
         /*
          * The last step likely did not land exactly on the
          * surface.  Backtrack to the surface and get the
          * coordinates of the new n-1 dimensional point.
          */
         double[] newPt = bactrackToSurface(lastSample.getPoint(), pt);
         /*
          * Create an output Sample object.  Use the latest
```

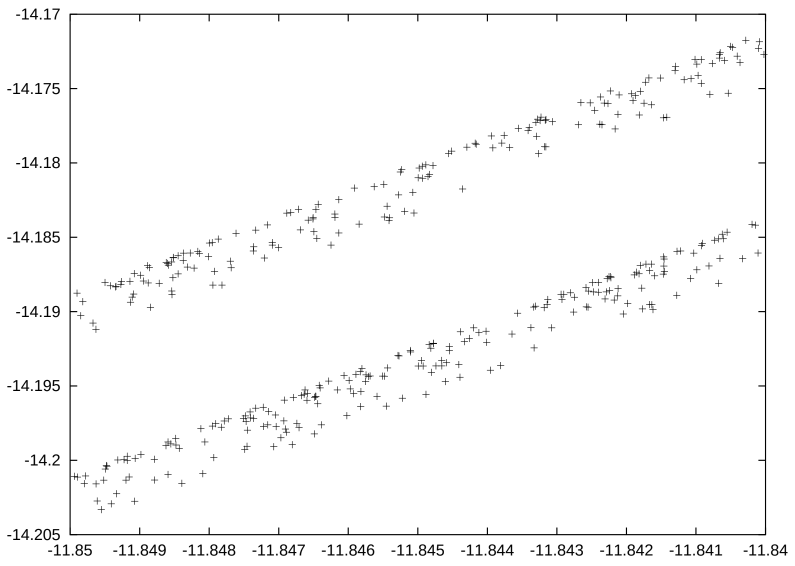

Figure 11: A closeup of a Poincaré map of the Lorenz attractr. The map is placed at $x = -8.5$. The horizontal and vertical axes of this graph are the $y$ and $z$ coordinates, respectively, where the continuous system pierced the surface of section.

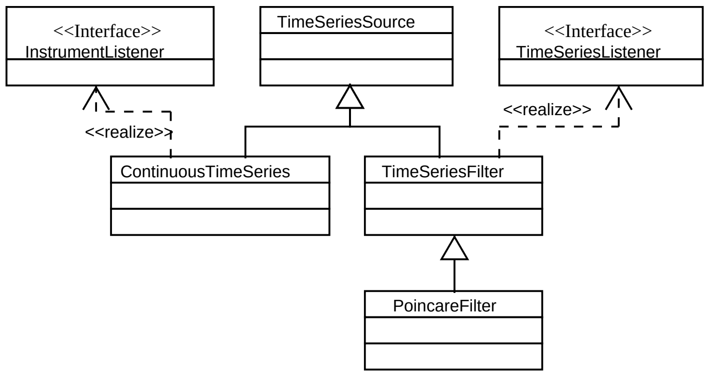

Figure 12: A Poincaré map may be implemented as a `TimeSeriesFilter`, converting a continuous time series to a discrete time series

```
          * scalar sample’s time as the embedded sample’s time.
          */
         Sample outSample =
            new Sample(newPt,inSample.getTime());
         notifyNewSample(new SampleEvent(this,outSample));
         lastPointNegative = false;
      }  }
      lastSample = inSample;
}  }
```

## 2.5 Find Periodic Behavior in Chaos

### 2.5.1 Context

The OGY method controls a chaotic system by exploiting the behavior of the system around the unstable periodic points of the chaotic attractor. A great strength of the OGY method is that it works without knowledge of the dynamical equations describing the system. However, the lack of an analytical solution means that fixed points must be located within chaotic activity empirically.

### 2.5.2 Problem

How can fixed points be determined from data sampled from a chaotic dynamical system?

### 2.5.3 Forces

- For an embedded time series, it is critical that the embedding space has enough dimensions to reconstruct the state trajectory of the chaotic system. If the embedding space has too few dimensions, the system’s trajectory will cross over itself leading to false identification of fixed points.
- Empirical detection of fixed points requires a period of uncontrolled sampling. The duration of the sampling period cannot be predicted from the outset, and it may be long.
- Finding periodic orbits in a continuous system may be expensive in terms of data and computation.
- The ergodic nature of chaotic systems guarantees that the system will approach all periodic points. Thus, it is guaranteed that there will be periodic points and that it is possible to get information about the system near an arbitrary periodic point.
- Finding periodic orbits in a continuous system may be considered equivalent to finding periodic points in a discrete system. A Poincaré section placed appropriately in the state space yields a discrete map where a point $x_i$ on the map represents the $i$th piercing of the map by the continuous trajectory.
- It is not clear in general how close a point and its successor have to be before they are considered to be close to a fixed point. The critical distance depends on the system in question.

### 2.5.4 Solution

A chaotic attractor has infinitely many periodic cycles[9]. For OGY control, it is necessary to find one or more of them by observing the behavior of the system. A $n$-cycle on a discrete map consists

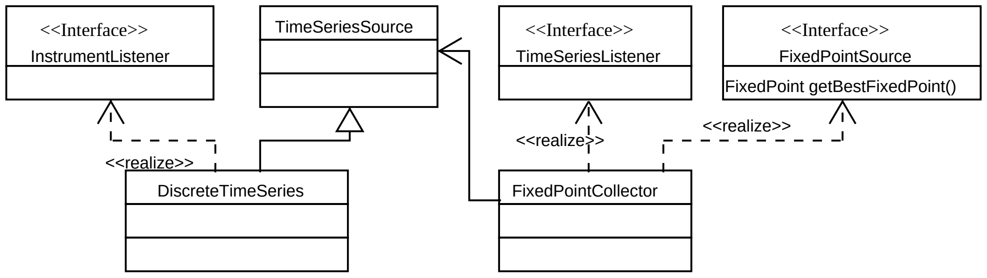

Figure 13: A fixed point collector may be a listener on the time series, estimating fixed point locations from pairs of successive points in the series.

of a sequence of $n$ points, $p_1, \ldots, p_n$ that repeats indefinitely on the map. Each point of a $n$-cycle of a discrete map $f$ is a fixed point of the map $f^n$, where

$$\begin{aligned} f^1(x) &= f(x) \\ f^2(x) &= f(f(x)) \\ &\vdots \end{aligned}$$

The location of a fixed point on a discrete map may be estimated from sampled data by observing pairs of successive points in the time series. Figure 13 shows how a fixed point collector fits into the example class hierarchy. The `FixedPointCollector` class implements the `TimeSeriesListener` interface, receiving a discrete time series of points from a `ChaoticSystem`. The `FixedPointCollector` implements the `FixedPointSource` interface which provides access to the best fixed point estimate currently available. The metric for fixed point quality is left to the `FixedPointCollector` implementation. Since system behavior changes smoothly with distance from the fixed point, a good candidate metric would be the distance from a point in the time series to its successor. Points that are closer together are considered to be better estimates of fixed points.

Figure 3 shows a trajectory of the Lorenz attractor that is estimated to be near a periodic trajectory–one that would repeat its course indefinitely. The depicted cycle would present itself as a fixed point on a surface of section placed at $x = 10$, and would be a 2-cycle on a surface of section placed at $x = -10$.

### 2.5.5 Example Code

```
public class FixedPointCollector
   implements TimeSeriesListener,FixedPointSource {
   private Point prevPoint = null;

   // ... (some code not shown)

   public FixedPoint getBestFixedPoint()
      {/* return best fixed point currently held */}

   public void readSample(SampleEvent e) {
      Point p = e.getSample().getPoint();
      double fixedness = distance(p, prevPoint);
```

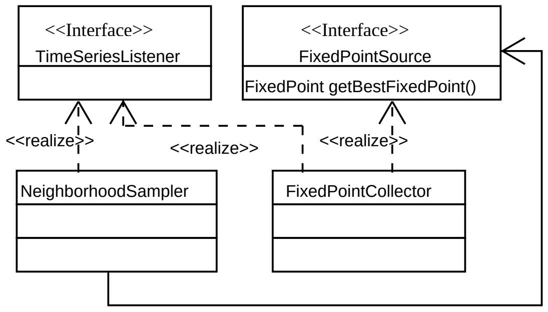

Figure 14: A neighborhood sampler listens to the time series and collects data points falling near fixed point estimates obtained from a fixed point source.

```
      if (fixedness < getMaxFixedness())
         storeFixedPoint(new FixedPoint(p,fixedness));
}  }
```

## 2.6 Sample the Fixed Point Neighborhood

### 2.6.1 Context

Once the location of a fixed point has been estimated, the OGY method must be able to characterize the behavior of the system around the fixed point. In order to accomplish this numerically, sample points a neighborhood around the fixed point must be collected.

### 2.6.2 Problem

How does one collect samples around a fixed point estimate for characterizing the locally linear behavior?

### 2.6.3 Forces

- In general, the optimal size of the neighborhood cannot be guessed without prior knowledge of the chaotic system.
- While sampling behavior around two different estimates of fixed points it might become apparent that the estimates are of the same fixed point. The data from the neighborhoods of the two estimates should be combined.
- A fixed point estimate as collected by the fixed point collector provides a rough estimate of the location of a possible fixed point on the attractor. By collecting samples of system activity around the estimate it is possible to more reliably characterize nearby system behavior.

### 2.6.4 Solution

Figure 14 depicts the `NeighborhoodSampler` class. It is a `TimeSeriesListener` so that it can obtain points from the discrete time series produced by the chaotic system. The `Neighborhood-`

`Sampler` also has a reference to a `FixedPointSource`, such as the `FixedPointCollector` of Section 2.5, to obtain estimates of fixed points to seed the sampling process. Each sample is an ordered pair $(x_i, x_{i+1})$, where $x_i$ is a point falling into the neighborhood of the fixed point and $x_{i+1}$ is the next point in the discrete time series from which $x_i$ was obtained.

It may not be clear at the outset what diameter of the neighborhood around a fixed point estimate is the right size to obtain control. One approach is to limit the number of neighborhood points collected around a fixed point to a size that is sufficient for subsequent calculations. Once the maximum number of points have been collected around a particular fixed point, the farthest point defines the diameter of the current neighborhood. When a closer point is encountered in the input stream it is added to the collection and the farthest point is removed. Through this process, the size of the neighborhood shrinks over time, automatically tuning to the system under observation.

As the `NeighborhoodSampler` collects data, it may improve its estimate of the location of the fixed point. For example, it is possible that while collecting samples around a fixed point estimate, a sample point falling into the region is also a better estimate of the fixed point. The `NeighborhoodSampler` should take into account improvements in the estimate of the fixed point location.

A chaotic system may have more than one fixed point on the discrete map representing its dynamics. The `NeighborhoodSampler` should be capable of sampling the neighborhoods of multiple fixed point estimates simultaneously. When sampling multiple fixed point neighborhoods simultaneously, the `NeighborhoodSampler` must determine from sampled data if two fixed point estimates refer to the same fixed point or if they represent the existence of two distinct fixed points.

### 2.6.5 Example Code

```
public class NeighborhoodSampler
   implements TimeSeriesListener
{
   FixedPointCollector fpCollector;
   Vector queues = new Vector();
   int numQueues;
   int queueSize;
   double[] prevPt = null;

   NeighborhoodSampler(FixedPointCollector fpc,int numQs,int qSize) {
      numQueues = numQs; queueSize = qSize; fpCollector = fpc;
   }

   public void sampleReceived(SampleEvent e) {
      Sample s = e.getSample();
      double[] pt = s.getPoint();

      manageQueues();
      if (prevPt != null && queues.size() > 0) {
         Iterator qIt = queues.iterator();
         NeighborhoodSample ns = new NeighborhoodSample(prevPt,pt);
         while (qIt.hasNext()) {
            NeighborQueue q = (NeighborQueue)qIt.next();
            try {
               q.put(ns);
```

```
                break; /* break if insert succeeds */
             }
             catch (NeighborQueue.TooFarAwayException e)
             {/*ignore-try next queue*/}
       }  }
       prevPt = pt;
    }

    private void manageQueues() {
       /* Create NeighborQueues using fixed point estimates
        * from fpCollector.  Make sure that two different queues
        * do not cover the same space (same fixed point region).
        */
       // ... (code not shown)
}  }

/**
 * A NeighborQueue stores samples in a double ended priority
 * queue.  If the maximum size of the queue is reached a new
 * sample must have higher priority than the lowest priority
 * sample in the queue.  When a new sample is inserted in this
 * condition, the lowest priority sample is removed and
 * discarded.  Priority is inversely proportional to the queue’s
 * fixed point estimate.
 */
public class NeighborQueue {
   /** thrown in put() if queue is full and new sample has lower
    *  priority than the lowest priority sample in the queue.
    */
   public class TooFarAwayException extends Exception {}

   /** Stores the neighborhood samples.  Priority is inverse
    *  of distance from this queue’s fixed point.
    *  @see #fp
    */
   private DePriQueue data;

   /** The point around which samples are collected */
   private FixedPointEstimate fp;

   // ... (not shown: constructor, accessors)

   /**
    * Put a new NeighborhoodSample in the queue if possible.
    * Each NeighborhoodSample contains a point and its successor.
    * @param item the NeighborhoodSample
   public void put(NeighborhoodSample item)
      throws TooFarAwayException {
      /* Confirm that the new sample has high enough priority to be
       * inserted into this queue.  If it does, then insert it.
       * If the new sample is a better fixed point estimate, then
       * designate new sample as the fixed point estimate and
       * reorder the queue.
       */
```

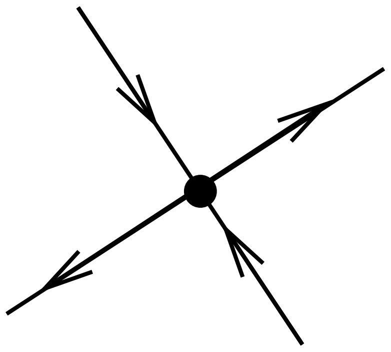

Figure 15: A controllable fixed point has stable regions, where the system tends to move towards the fixed point, and unstable regions, where the system tends to move away. The stable and unstable regions are identified from the linear approximation of dynamics near the fixed point. OGY control attempts to keep the system in the stable region of the fixed point.

```
      // ... (code)
}  }
```

## 2.7 Linearize Dynamics near the Fixed Point

### 2.7.1 Context

OGY control is carried out around fixed points that have both stable and unstable regions. When the system state is in the stable region it will tend to move towards the fixed point in the next iteration. In the unstable region, the system tends to move away from the fixed point. Values derived from a linear approximation of dynamics near a fixed point are used by the OGY method to establish and maintain control of the system. Figure 15 illustrates the type of information derived from linearization.

The theoretical process for estimating linear dynamics has been documented in chaos control literature[1]. It is not clear that a straightforward implementation of the process is the best for all situations. The component will be implemented differently depending upon the criteria placed upon the estimates. For example, there is generally a tradeoff between computational efficiency and data efficiency. Depending upon the target computation environment, one or the other of these criteria may have to be sacrificed.

### 2.7.2 Problem

How can the estimation of linear dynamics near a fixed point be carried out in a way that allows for variation in implementation that is transparent to the rest of the system?

### 2.7.3 Forces

- Assuming no analytical solution to the system in question, the local dynamics must be estimated from sampled data.

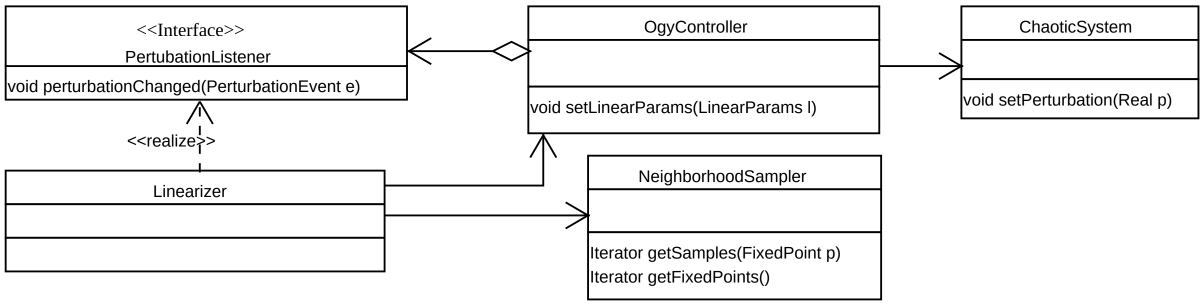
<<Interface>>
PertubationListener
void perturbationChanged(PerturbationEvent e)
<<realize>>
Linearizer
OgyController
void setLinearParams(LinearParams l)
ChaoticSystem
void setPerturbation(Real p)
NeighborhoodSampler
Iterator getSamples(FixedPoint p)
Iterator getFixedPoints()

Figure 16: The dynamical behavior near a fixed point is estimated by the `Linearizer` class. The `Linearizer` receives sample points from the `NeighborhoodSampler`. The control parameter setting is stored with the linearization around each fixed point.

- The optimal size of the neighborhood for which a linear approximation is appropriate is system dependent, and not necessarily known at the outset.
- There is a tension between the size of the linearized neighborhood and the ability to control. If the neighborhood is too large, the linearization will be less useful. If the neighborhood is too small it will take too long for the system to fall into the controllable region.

### 2.7.4 Solution

The `Linearizer` component depicted in Figure 16 encapsulates the function of linearizing the dynamics of the system near a fixed point. The `Linearizer` gets fixed point neighborhood sample data from the `NeighborhoodSampler`. The `getFixedPoint()` method of `NeighborhoodSampler` returns an `Iterator` of `FixedPoint` objects that represent the fixed point estimates for which neighborhood information is currently being sampled. The sampled points around a particular fixed point may be retrieved with the `getSamples(FixedPoint)` method, which returns an iterator of `SamplePair`objects. Each `SamplePair` object contains two `Points`, representing a point falling in the neighborhood of the fixed point and the next iterate of the discrete map.

There is two-way communication between the `OgyController` and the `Linearizer`. Communication from the `OgyController` is performed indirectly through the `PerturbationListener` interface which is implmented by the `Linearizer`. Whenever the `OgyController` changes the perturbation value of the `ChaoticSystem`, it broadcasts an event to its `PerturbationListeners` notifying them of the change. The `LinearParams` class encapsulates the linear dynamics information in a form useful to the `OgyController`. Because the perturbation parameter can change the location and the local behavior of a fixed point, the `Linearizer` stores the current perturbation value with the `LinearParams` object created for a fixed point. To obtain perturbation parameter change information, the `Linearizer` is a `PerturbationListener`. Once a linear estimate has been produced, the `Linearizer` provides the estimate to the `OgyController` through its `receiveLinearParams()` method.

### 2.7.5 Example Code

```
public class Linearizer implements PerturbationListener {
   private OgyController controller;
   private NeighborhoodSampler sampler;
   private double perturb;

   public Linearizer(OgyController c,NeighborhoodSampler s) {
      controller = c; sampler = s; }

   /**
    * Changing the perturbation changes the dynamical behavior, so
    * get the samples collected under the previous perturbation
    * value and estimate fixed point neighborhood dynamics.
    * @param e event notifying of change in perturbation value.
    *        The PerturbationEvent object contains the new value
    *        of the perturbation parameter.
    * @see java.util.EventObject
    */
   public void perturbationChanged(PerturbationEvent e) {
      double newPerturb = e.getPerturbation();
      Iterator fpIt = sampler.getFixedPoints();
      while (fpIt.hasNext()) {
         FixedPoint fp = (FixedPoint)fpIt.next();
         Iterator sampleIt = sampler.getSamples(fp);
         Vector v = new Vector();
         while (sampleIt.hasNext()) v.add(sampleIt.next());
         LinearParams lp = linearizeNeighborhood(fp,v,perturb);
         controller.setLinearParams(lp);
      }
      perturb = newPerturb;
   }

   /** Calculates values representing the linear characterization
    * of behavior around the fixed point based on samples.
    * @return the parameters
    */
   private LinearParams linearizeNeighborhood(
      FixedPoint fp,Vector v,double perturb) {
      LinearParams lp = new LinearParams();
      // ... (calculate parameters and populate lp with them)
      return lp;
}  }
```

## 2.8 Track Fixed Point Migration

### 2.8.1 Context

When the system falls within the controllable region of a fixed point, the `OgyController` perturbs the `ChaoticSystem` control parameter in such a way that system state will tend to fall in the controllable region of the fixed point on the next iteration. The `OgyController` obtains information about the location of the stable and unstable regions around a fixed point from the `Linearizer`for a particular setting of the perturbation parameter. By perturbing the `ChaoticSystem`'s control paramter When the chaotic system state lands near a fixed point for which the `OgyController` has control information, the `OgyController` attempts to keep the state in the neighborhood of

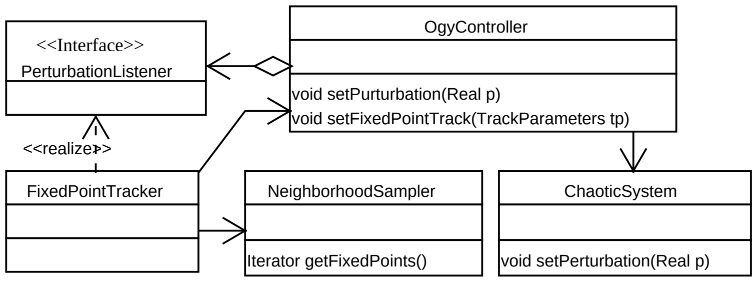
<<Interface>>
PerturbationListener
OgyController
void setPurturbation(Real p)
void setFixedPointTrack(TrackParameters tp)
<<realize>>
FixedPointTracker
NeighborhoodSampler
Iterator getFixedPoints()
ChaoticSystem
void setPerturbation(Real p)

Figure 17: The `FixedPointTracker` obtains fixed point estimate values from the `NeighborhoodSampler`, then contacts the `OgyController` to change the control parameter value for a new round of sampling. After finding the fixed points of the perturbed system, fixed point migration parameters are calculated and sent to the `OgyController`.

the fixed point on succeeding iterations. If the state is in the unstable region of the fixed point the state will move away from the fixed point. In such a case, the `OgyController` will perturb the `ChaoticSystem` so that the current state is in the stable region of the perturbed system's fixed point, thereby achieving the desired control.

### 2.8.2 Problem

How can one predict the movement of a fixed point under perturbation?

### 2.8.3 Forces

- The fixed point is assumed to move smoothly with respect to small changes to the control parameter.
- Chaotic systems can change drastically with minute perturbations. For example, if the parameter is moved past a bifurcation point for the system, the fixed point may become a two-cycle. It is better to avoid such situations.
- The notion of what is a small perturbation change varies with each chaotic system. The range of safe parameter change values must be determined through experience with the chaotic system of interest.

### 2.8.4 Solution

Once a fixed point is located in the unperturbed system, a small perturbation is applied to the system and the fixed points of the perturbed system are located. To track the movement of the fixed point of interest, it is matched with the nearest fixed point of the perturbed system. To characterize the movement of the fixed point under perturbation, a linear interpolation between the two fixed point locations is calculated. The OGY method uses this linear interpolation along with the information provided by the `Linearizer` to calculate the amount of perturbation to apply.

Figure 17 illustrates class relationships for the `FixedPointTracker` class, which calculates parameters for estimating the migration of fixed points under control parameter perturbation. The

`FixedPointTracker` obtains fixed point estimates of the unperturbed system from the `Neigh-borhoodSampler`, then requests that the `OgyController` apply a perturbation to the `Chaotic-System`. It implements the `PerturbationListener` interface to ensure that the starting After a period of sampling the perturbed system, the FixedPointTracker obtains new fixed point estimates from the NeighborhoodSampler, matches them up with fixed points of the unperturbed system, then calculates linear interpolation parameters for estimating the movement of each fixed point with respect to the control parameter. These parameters are communicated to the `OgyController` in a `TrackParameters` object. The `TrackParameters` class encapsulates the linear interpolation parameters in the form used in the `OgyController`.

### 2.8.5 Example Code

```
public class FixedPointTracker implements PerturbationListener {
   private OgyController controller;
   private NeighborhoodSampler sampler;
   private double perturb;
   /**
    * Stores fixed point estimates along with the perturbation
    * values at which they were encountered.
    */
   private Collection fpHistory;

   public FixedPointTracker(OgyController c,NeighborhoodSampler s) {
      controller = c; sampler = s; }

   /**
    * Get the fixed point estimates from the NeighborhoodSampler.
    * These pertain to the previous perturbation value.  If possible,
    * calculate fixed point migration parameters and pass them
    * to the OgyController.
    * @param e event notifying of change in perturbation value.
    *        The PerturbationEvent object contains the new value
    *        of the perturbation parameter.
    * @see java.util.EventObject
    */
   public void perturbationChanged(PerturbationEvent e) {
      double newPerturb = e.getPerturbation();
      Iterator fpIt = sampler.getFixedPoints();
      while (fpIt.hasNext()) {
         TrackParameters tp;
         tp = trackFp((FixedPoint)fpIt.next(),perturb);
         controller.setFixedPointTrack(tp);
         /* HistoryItem is a class that stores a FixedPoint
          * value and a perturbation value.
          */
         fpHistory.add(new HistoryItem(fp,perturb));
      }
      perturb = newPerturb;
   }

   /**
    * Determines the movement of a fixed point with respect
```

```
     * to the perturbation parameter.
     * @return an object characterizing the movement of the
     *         fixed point.
     */
    private TrackParameters trackFp(FixedPoint fp,double p) {
       TrackParameters tp = new TrackParameters();
       /* Get all fixed points from fpHistory for a perturbation
        * value near p.  Match fp with the nearest of these fixed
        * points.  Populate tp with parameters estimating the
        * movement of fp with respect to p.
        */
       // ... (code)
       return tp;
 }  }
```

## 3 Conclusion

The pattern language described here provides the basis for a flexible software architecture for working with chaotic systems. Since OGY control operates on general principles of chaotic systems, the language includes patterns that separate application specific code from the more general code applicable to all chaotic dynamical systems. Having accomplished such a separation, it is possible to develop general OGY control code that may be applied without modification to a variety of chaotic systems. In doing so it becomes possible to study the OGY code independent of the system being controlled.

The OGY control procedure has been further subdivided into data transformation, information gathering, and processing patterns. The data transformation patterns, Time Series Data Filter; Embed the Time Series; and Apply a Poincaré Map, serve to separate the details of a particular chaotic system from the core of the OGY method. The information gathering patterns, Find Periodic Behavior in Chaos; and Sample the Fixed Point Neighborhood, provide the raw information that the OGY method requires for control. The processing patterns, Linearize Dynamics near the Fixed Point; Track Fixed Point Migration; and Control a Chaotic System, take the collected information, compute the parameters for OGY control, and initiate the perturbations to the chaotic system to establish control.

The relative independence of each of these patterns allows for loose coupling between the components implementing them. This loose coupling in turn allows for easy modification or replacement of components, which is advantageous in an experimental environment where different techniques for implementing a pattern are to be tested. In addition to facilitating component replacement, loose coupling allows flexibility in the linkage between components leading to possible parallel execution.

### 3.1 Future Work

The pattern language described here lays the foundation for the development of a fully autonomous OGY controller. To reach this goal, further elaboration of the Embed the Time Series; Apply a Poincaré Map; and Track Fixed Point Migration patterns is necessary since these patterns state what needs to be done but are vague on how to actually do it. Each of these patterns has aspects which tend to reduce its tractability in an unsupervised mode.

In the case of Embed the Time Series, two variables must be calculated. First, the number of dimensions in which to embed the time series must be calculated. It is necessary to provide enough dimensions so that the state space behavior is faithfully reconstructed. While on the one hand the problem of embedding is to make the number of dimensions sufficiently large, it is also important to minimize the number of dimensions as much as possible to minimize the complexity of later calculations. In a noiseless environment, it would be sufficient to choose the lowest number of dimensions in which the embedded trajectory does not cross itself. Since noise is an ever present component of real world systems and a chaotic trajectory will pass arbitrarily close to itself, an embedded trajectory may appear to cross itself due to noise interference. Therefore a calculation of embedding must accept some amount of trajectory intersection. The second variable affecting the embedding is the time delay between coordinates of the embedded system. This is the $T$ value of Equation 4. It has the effect of warping the reconstructed trajectory, and a well chosen $T$ can aid in the calculation of embedding dimension.

A well placed Poincaré map makes the job of locating fixed points easier. It should be placed so that it is transverse to the flow and so that the flow passing through it contains unstable periodic trajectories that may be used for control.

Tracking fixed point migration with respect to parameter perturbations involves application of a small perturbation to the system, locating the fixed points of the perturbed system, and matching them up with those of the unperturbed system. It is necessary for an unsupervised system to calculate what is meant by “small.” One may base the size of the perturbation on the diameter of the sampled fixed point neighborhood. Depending upon the sensitivity of a particular system to the perturbation parameter, other heuristic rules may be more appropriate. The OGY method tends to assume that manipulation of the perturbation parameter merely deforms the chaotic attractor. However, a perturbation may result in bifurcation of a fixed point or more profound change in dynamical behavior. An unsupervised controller must be able to take these factors into account while tuning the range of perturbations to the chaotic system.

With the advent of computing clusters built from commodity PCs, parallel execution is an increasingly viable approach to accomplishing all necessary processing in the short time interval between samples. Using the mechanism introduced in Time Series Data Filter for inserting processing steps in the chain of communication, it is possible to provide a type of filter that communicates between distributed processes. By using technologies such as CORBA or Java Remote Method Invocation, procedure calls across a network link may be accomplished transparently through the interface of the target class. Components need not be aware of the distributed nature of their environment.

## 4 Acknowledgements

The author wishes to thank Jeff McGough of SDSMT for many stimulating discussions and helpful suggestions regarding this paper. Thanks to Troy McVay for nice cyclic behavior examples on the Lorenz attractor.

## References

[1] S. Boccaletti, C. Grebogi, Y.-C. Lai, H. Mancini, and D. Maza. The control of chaos: Theory and applications. *Physics Reports*, 329:103–197, 2000.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[3] J. Guckenheimer and P. Holmes. *Nonlinear Oscillations, Dynamical Systems and Bifurcations of Vector Fields*, volume 42 of *Applied Mathematical Sciences*. Springer Verlag, New York, 1983.

[4] M. Henon. On the numerical computation of Poincaré maps. *Physica D*, 5:412–414, 1982.

[5] E. N. Lorenz. Deterministic nonperiodic flow. *Jorunal of the Atmospheric Sciences*, 20:130–141, March 1963.

[6] E. Ott, C. Grebogi, and J. A. Yorke. Controlling chaos. *Physical Review Letters*, 64(11):1196–1199, 1990.

[7] G. T. Stubbendieck. *Numerical Methods for the Control of Chaos*. PhD thesis, Texas Tech University, Lubbock, TX, 1993.

[8] F. Takens. Detecting strange attractors in turbulence. In D. Rand and L.-S. Young, editors, *Dynamical systems and turbulence*, pages 366–381, Berlin, 1981. University of Warwick, Springer-Verlag.

[9] J. A. Yorke and T. Y. Li. Period three implies chaos. *Amer. Math. Monthly*, 82:985–992, 1975.