# Class Design for Siena 4

Krists Boitmanis

September 8, 2009

## 1. Introduction

The purpose of this document is to give an overview of the C++ code for Siena 4. Some notes:

- Names in the code are indicated as *name*.

- Typewriter font is used for file names, like `FileName.cpp`.

- Normally, a class named *SomeName* is declared in the file `SomeName.h` and implemented in the file `SomeName.cpp`.

- Camel case is used for names of classes, methods, and variables. More specifically, all names but class names start with a lower case letter, and when the name consists of several words, each of the subsequent words starts with an upper case letter. Examples:

  - *StatisticCalculator* – a class name
  - *calculateNetworkRateStatistics* – a method name
  - *observationCount* – a variable name

- The names of pointer variables start with a 'p' followed by an upper case letter, like *pNetworkData*.

- Similarly, the names of reference variables start with an 'r', like in the following declarations of a copy constructor and an assignment operator:

```
Network(const Network & rNetwork);
Network & operator=(const Network & rNetwork);
```

- Moreover, the names of instance variables are prepended with an 'l', e.g.

  - int *lobservationCount* – an instance variable of a non-pointer type,
  - *Model* \* *lpModel* – an instance variable of a pointer type.

## 2.   Library Overview

The code is organised into libraries that correspond to the directory names under `src`. The libraries are briefly explained in the following list in the order of their dependencies, namely, no library depends on any other library listed after it.

- `utils` – contains various general purpose classes and functions.

- `network` – contains the classes *Network* and *OneModeNetwork* for storing and manipulating directed two-mode and one-mode networks, respectively, as well as supporting classes like various iterators over ties in a network.

- `data` – contains everything related to the observed data with the class *Data* being the main container storing the observations of network and behavior variables, covariates, etc.

- `model` – contains the tools for specifying and simulating actor-oriented models. The *Model* class can be used to specify an actor-oriented model, which can be subsequently simulated with respect to a specific *Data* object by using an instance of the *EpochSimulation* class. Finally, the *StatisticCalculator* class provides the means for calculating the observed or simulated statistics for effects in the model.

In the next sections, each library is explained in some detail.

## 3. Utils

- `Utils.h` provides miscellaneous utility functions, macros, and classes.

- `Random.h` provides functions for drawing random numbers.

- The class *NamedObject* acts as a base class for anything that has a name.

- The singleton class *SqrtTable* is used throughout the system to calculate square roots of integers efficiently. Once the root of an integer is calculated, it is stored in a table for later reuse.

  Example:

  ```
  SqrtTable * pTable = SqrtTable::instance();
  double root2 = pTable->sqrt(2);
  ```

## 4. Networks

### 4.1 *Network* class

The *Network* class implements directed two-mode networks with valued ties. The number of tie senders $n$ and the number of tie receivers $m$ have to be provided at the construction of a network. Technically, the *Network* class can be used for storing one-mode networks as well, in which case $n = m$, however, the derived class *OneModeNetwork* is recommended for this purpose.

### 4.1.1 Internal Data Representation

The outgoing ties of each sender $i$ are stored in an STL (C++ Standard Template Library) *map*. The alters of non-zero ties from $i$ act as keys in the map, and the tie values are stored in the map as values corresponding to those keys. The maps themselves are stored in an array *lpOutTies*, hence the value of a tie $(i, j)$ can be accessed as *lpOutTies*$[i][j]$. Similarly, an array of maps *lpInTies* is used to store the incoming ties of all receivers.

While being much more memory efficient than storing the adjacency matrix of a network explicitly, this representation still provides fast queries of tie values – the number of steps necessary to retrieve the value of a tie $(i, j)$ is

proportional to the logarithm of the out-degree of $i$. Also, this representation directly provides a fast and convenient way of iterating over all non-zero ties of an actor in the increasing order of the alters.

The users of the *Network* class should not care about its internal representation, though, and use the public interface methods, to which we now turn.

### 4.1.2 Interface

Some of the more important methods of the *Network* class are explained in the following list.

- *Network*(`int n`, `int m`) – constructs a network with $n$ senders, $m$ receivers, and no ties.

- `int` *n*() – returns the number of actors in the set of tie senders.

- `int` *m*() – returns the number of actors in the set of tie receivers.

- `void` *setTieValue*(`int i`, `int j`, `int v`) – sets the specified value of the tie $(i, j)$.

- `int` *tieValue*(`int i`, `int j`) – returns the value of the tie $(i, j)$.

- *TieIterator* *ties*() – returns an iterator over all ties of the network. The ties are ordered according to their senders from smallest to largest, and the ties from the same sender are ordered according to their receivers.

  Usage:

  ```
  TieIterator iter = pNetwork->ties();

  while (iter.valid())
  {
    int ego = iter.ego();
    int alter = iter.alter();
    int value = iter.value();

    cout << "The tie from " << ego << " to " << alter <<
  ```

4

```
      " has a value " << value << endl;

    // Move on to the next tie
    iter.next();
  }
```

- *IncidentTieIterator outTies*(int *i*) – returns an iterator over the outgoing ties of actor *i* in the increasing order of the alters.

  Usage:

```
IncidentTieIterator iter = pNetwork->outTies(i);

while (iter.valid())
{
  int alter = iter.actor();
  int value = iter.value();

  cout << "The tie from " << i << " to " << alter <<
    " has a value " << value << endl;

  // Move on to the next tie
  iter.next();
}
```

- *IncidentTieIterator inTies*(int *i*) – returns a similar iterator over the incoming ties of actor *i*.

- int *outDegree*(int *i*) – returns the number of non-zero outgoing ties for actor *i*.

- int *inDegree*(int *i*) – returns the number of non-zero incoming ties for actor *i*.

## 4.2  Iterators

We have already seen the usage of supporting classes *TieIterator* and *IncidentTieIterator*. Another iterator class, namely *CommonNeighborIterator*, provides convenient means for iterating over actors that are common to two instances of *IncidentTieIterator*. For example, one can iterate over reciprocated ties of an actor in a one-mode network as follows:

```
CommonNeighborIterator iter(pNetwork->inTies(i),
  pNetwork->outTies(i));

while (iter.valid())
{
  int neighbor = iter.actor();
  cout << "There is a reciprocated tie from " << i <<
    " to " << neighbor << endl;

  // Move on to the next neighbor
  iter.next();
}
```

## 4.3 *OneModeNetwork* class

A one-mode network can be thought of as a special case of a two-mode network, where the set of tie senders coincides with the set of tie receivers. Hence, the *OneModeNetwork* class is derived from the *Network* class. In addition to the interface inherited from the base class, it maintains the number of reciprocated ties per each actor and provides various methods for counting two-paths in a network. A non-exhaustive list of methods follows:

- *OneModeNetwork*(`int` *n*, `bool` *loopsPermitted*) – constructs an empty network for $n$ actors. Ties from actors to themselves (which are called loops) can be explicitly forbidden by providing the value `false` as the second argument.

- `int` *reciprocalDegree*(`int` *i*) – returns the number of reciprocated ties of actor $i$.

- *CommonNeighborIterator* *reciprocatedTies*(`int` *i*) – returns an iterator over all reciprocated ties of actor $i$. The usage is similar to that of the method *Network*::*outTies*(`int` *i*) except that the class *CommonNeighborIterator* does not have a *value*() method. In a sense, the network is treated as a boolean network, where the existence/non-existence of ties is of interest, but not the actual values of ties.

- int *twoPathCount*(int *i*, int *j*) – returns the number of two-paths from actor *i* to actor *j*.

## 5. Observed Data

The `data` library contains classes for storing the observed data subject to actor-oriented modeling. The entire set of observed data for a Siena project (or one group in a multi-group project) should be stored in an instance of the *Data* class. It owns collections of more specific data objects like those storing the observed data for dependent network variables, constant actor covariates, etc. This section will guide you step-by-step through the process of populating a *Data* object. To begin with, an instance of the *Data* class should be created:

```
int observationCount = 4;
Data * pData = new Data(observationCount);
```

Note that the number of observations has to be provided in the very beginning.
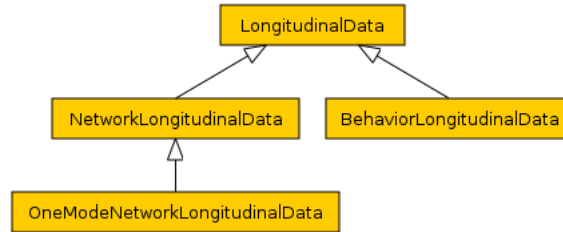
### 5.1  Actor Sets

There may be several sets of actors involved, like a set of students, a set of teachers, and a set of courses, therefore, when specifying dependent variables and covariates, one should be explicit about the relevant actor sets. Each set of actors is represented in the system by an instance of the *ActorSet* class, which can be created with the factory method *Data*::*createActorSet* by providing the name of the set and the number of actors in it.

```
const ActorSet * pStudents =
  pData->createActorSet("students", 50);
const ActorSet * pCourses =
  pData->createActorSet("courses", 20);
```

## 5.2 Observed Data for Dependent Variables

There is a small hierarchy of classes for storing the observed data for dependent variables.



The base class *LongitudinalData* is not supposed to be created directly, however, it provides some useful methods applicable to observed data of all types of dependent variables:

- `const` *ActorSet* `*` *pActorSet*() – returns the set of actors this dependent variable is defined for.

- `int` *n*() – returns the size of the set of actors.

- `int` *observationCount*() – returns the number of observations that can be stored in this data object.

- `bool` *upOnly*(`int` *period*) – returns if only upward changes are observed in the given period.

- `void` *upOnly*(`int` *period*, `bool` *flag*) – stores the indicator if only upward changes are observed in the given period. Note that the *LongitudinalData* class provides just the storage of these flags, and their values have to be computed elsewhere. They are currently passed in from the R part of the system.

- *downOnly* – similar accessor methods for flags indicating that only downward changes are observed.

The derived classes of *LongitudinalData* are created with the factory methods of the *Data* class by providing the name of the variable and the corresponding actor set (or both the set of tie senders and the set of tie receivers in the case of two-mode networks):

8

- one-mode networks

  ```
  OneModeNetworkLongitudinalData * pFriendshipData =
    pData->createOneModeNetworkData("friendship",
      pStudents);
  ```

- two-mode networks

  ```
  NetworkLongitudinalData * pCourseSelectionData =
    pData->createNetworkData("courses",
      pStudents,
      pCourses);
  ```

- behavior

  ```
  BehaviorLongitudinalData * pAverageGradeData =
    pData->createBehaviorData("grade", pStudents);
  ```

### 5.2.1  Two-Mode Network Variables

**Network Structure.**   The main purpose of the *NetworkLongitudinalData* class is to store the values of network ties in each of the observations as well as indicators about whether a tie value is missing or structurally determined. This can be done with the following setter methods:

- void *tieValue*(int $i$, int $j$, int *observation*, int *value*) – stores the value of the tie between the given actors at the given observation,

- void *missing*(int $i$, int $j$, int *observation*, bool *flag*) – stores if the value of the tie between the given actors is missing at the given observation,

- void *structural*(int $i$, int $j$, int *observation*, bool *flag*) – stores if the value of the tie between the given actors is structurally determined at the given observation.

There is a corresponding set of methods for accessing the stored values:

- int *tieValue*(int $i$, int $j$, int *observation*)

9

- bool *missing*(int *i*, int *j*, int *observation*)

- bool *structural*(int *i*, int *j*, int *observation*)

Internally, this information is stored in three arrays of networks, namely, *lnetworks* for the observed tie values, *lmissingTieNetworks* for indicators of missing ties, and *lstructuralTieNetworks* for storing the indicators of structurally determined ties. To enable effective iteration over all ties, the class *NetworkLongitudinalData* provides constant access to these networks via the following methods:

- const *Network* * *pNetwork*(int *observation*) – returns the network of observed values at the given observation,

- const *Network* * *pMissingTieNetwork*(int *observation*) – returns the network of missing tie indicators for the given observation,

- const *Network* * *pStructuralTieNetwork*(int *observation*) – returns the network of structural tie indicators for the given observation.

For example, the following code snippets demonstrate two ways of iterating over the selected courses of a certain student $i$, but the second snippet is much more efficient.

*Example 1*

```
for (int j = 0; j < pCourses->n(); j++)
{
  if (pCourseSelectionData->tieValue(i, j, observation) != 0)
  {
    cout << "The student " << i <<
      " has selected the course " << j << endl;
  }
}
```

*Example 2*

```
const Network * pNetwork =
  pCourseSelectionData->pNetwork(observation);

IncidentTieIterator iter = pNetwork->outTies(i);

while (iter.valid())
{
  int j = iter.actor();
  cout << "The student " << i <<
    " has selected the course " << j << endl;
  iter.next();
}
```

**Calculating Properties.** Once a network data object has been populated, it is important to call the *calculateProperties* method that computes important properties of the observed data, which are used by some effects during model simulations:

```
pCourseSelectionData->calculateProperties();

cout << "The average number of courses per student is " <<
  pCourseSelectionData->averageOutDegree() << endl;
cout << "The average number of students " <<
  "attending a course is " <<
  pCourseSelectionData->averageInDegree() << endl;
```

**Other Methods.**

- void *maxDegree*(int *degree*) – if there is a restriction on the maximum number of outgoing ties an actor can have, it has to be specified by using this method.

### 5.2.2   One-Mode Network Variables

Since one-mode networks are a special kind of two-mode networks, the class *OneModeNetworkLongitudinalData* derives from *NetworkLongitudinalData* and

inherits all its methods. The derived class provides some additional
methods, though, which are applicable to one-mode networks only:

- void *symmetric*(`bool` *flag*) – stores if the network is symmetric,

- void *balanceMean*(`double` *value*) – stores the centering constant for the
  balance effect.

Some of the information that is simply stored in the
*OneModeNetworkLongitudinalData* class could be computed by the class itself,
but since this information is already computed in R, we save the effort of
computing it again, and simply pass the values in from the R part of the
system.

### 5.2.3 Behavior Variables

Analogous to the data class for network variables, the class
*BehaviorLongitudinalData* stores the observed values of a certain behavior
variable and various properties computed from the observed data.

**Observed Values.** The observed values or indications about missing
values can be stored with the following methods:

- void *value*(`int` *observation*, `int` *actor*, `int` *value*)

- void *missing*(`int` *observation*, `int` *actor*, `bool` *missing*)

The stored values can be accessed with these methods:

- int *value*(`int` *observation*, `int` *actor*)

- bool *missing*(`int` *observation*, `int` *actor*)

Internally, the observed values are stored in an $M \times N$ integer array *lvalues*
and the missingness indicators are stored in an $M \times N$ boolean array
*lmissing*, where $M$ is the number of observations and $N$ is the number of
actors in the corresponding actor set. A read-only access to a whole row of
the matrix of values (namely, the values of all actors in a single
observation) is provided by the method

- const int * *values*(`int` *observation*).

**Calculating Properties.** Again, as soon as the observed values and missingness indicators are stored, and before the data object can be used for model simulations, the method *calculateProperties* should be called:

```
pAverageGradeData->calculateProperties();

cout << "The average grades of students range from " <<
  pAverageGradeData->min() << " to " <<
  pAverageGradeData->max() << endl;
```

The method *calculateProperties* calculates some properties of the observed data, which can be subsequently queried with the following methods:

- `int` *min*() – returns the smallest observed non-missing value,

- `int` *max*() – returns the largest observed non-missing value,

- `int` *range*() – returns the range of observed non-missing values, which is simply the difference between the maximum and the minimum values,

- `double` *overallMean*() – returns the overall mean of the observed non-missing values defined as $\frac{1}{M} \sum_{k=1}^{M} \frac{\sum_{i \in A_k} v_i^k}{|A_k|}$, where $v_i^k$ is the observed behavior for actor $i$ and the observation $k$ and $A_k$ is the set of actors with non-missing values at the observation $k$.

**Other Methods.**

- `void` *similarityMean*(`double` *similarityMean*) – stores the mean similarity of actor behavior values over all observations, which is calculated in R.

- `double` *similarity*(`double` *a*, `double` *b*) – returns the centered similarity score for the given values $a$ and $b$. The similarity score is defined as $1 - \frac{|a-b|}{\Delta}$, where $\Delta$ is the observed range of the behavior variable, and the mean similarity is subtracted from this expression to obtain the centered similarity score.

## 5.3 Covariates

The classes for storing actor covariates and dyadic covariates are organised in two separate hierarchies:



Since the usage of covariate classes is very similar to that of observed data for behavior variables, we will just briefly list the main methods for working with covariates.

The covariate data classes can be created by the factory methods of the *Data* class:

- *ConstantCovariate* \* *createConstantCovariate*(...)

- *ChangingCovariate* \* *createChangingCovariate*(...)

- *ConstantDyadicCovariate* \* *createConstantDyadicCovariate*(...)

- *ChangingDyadicCovariate* \* *createChangingDyadicCovariate*(...)

The name of the covariate and one (for actor covariates) or two (for dyadic covariates) actor sets should be provided as parameters for these methods.

### 5.3.1 Actor Covariates

The base class *Covariate* for actor covariates provides the following methods:

- void *range*(double *range*) – stores the observed range of the covariate. Unlike the observed data for behavior variables, it is not computed by the covariate class itself, but passed in from R.

- double *range*() – returns the observed range of the covariate.

- *similarityMean*(...) and *similarity*(...) – these methods behave precisely as the corresponding methods in the *BehaviorLongitudinalData* class.

The derived classes *ConstantCovariate* and *ChangingCovariate* provide appropriate methods *value*(...) and *missing*(...) for storing and retrieving the values of the covariate and the missingness indicators, respectively. Internally, the data is stored in one-dimensional or two-dimensional $N \times M$ arrays *lvalues* and *lmissing*, where $N$ is the number of actors in the respective set and $M$ is the number of observations.

### 5.3.2  Dyadic Covariates

The interface of the dyadic covariates is very similar to that of actor covariates and should not present any difficulties. The base class *DyadicCovariate* provides access to both relevant sets of actors (methods *pFirstActorSet*() and *pSecondActorSet*()) and a storage of the mean value of the covariate that is passed in from R (*mean*(...) methods).

Again, the values of the covariate and their missingness indicators can be stored and accessed with methods *value*(...) and *missing*(...), which are defined in the derived classes *ConstantDyadicCovariate* and *ChangingDyadicCovariate*.

The internal representation of the values is tricky, though. Since storing a sparse $N_1 \times N_2$ (the sizes of both involved actor sets) matrix explicitly is expensive in terms of memory, we store each row of the matrix as an instance of *map*<int, double> that maps each column index with a non-zero value in that row to the respective value. The matrix of missings has only 0 and 1 as its entries, so an instance of *set*<int> is more appropriate for storing the column indices with missings in a certain row. Array of such maps or sets (like *lpRowValues* and *lpRowMissings* in *ConstantDyadicCovariate*), one for each row, then represents the whole matrices. Such a representation is not only space-efficient, but also suitable for fast iteration over non-zero values in a certain row. To facilitate an equally fast iteration over non-zero values in a certain column, we maintain analogous structures for columns in *lpColumnValues* and *lpColumnMissings*. The following methods in *ConstantDyadicCovariate* exploit this internal representation to provide fast iterators over non-zero non-missing values of in a given row or column:

- *DyadicCovariateValueIterator rowValues*(int *i*)

- *DyadicCovariateValueIterator columnValues*(int *j*)

*Example*

The following example multiplies the row $i$ with the column $j$.

```
DyadicCovariateValueIterator rowIter =
  pCovariate->rowValues(i);
DyadicCovariateValueIterator columnIter =
  pCovariate->columnValues(j);

double product = 0;

while (rowIter.valid() && columnIter.valid())
{
  if (rowIter.actor() < columnIter.actor())
  {
    rowIter.next();
  }
  else if (rowIter.actor() > columnIter.actor())
  {
    columnIter.next();
  }
  else
  {
    product += rowIter.value() * columnIter.value();
    rowIter.next();
    columnIter.next();
  }
}
```

Finally, we should note that the *ChangingDyadicCovariate* class has the same functionality as discussed above for constant dyadic covariates, except that the methods expect the observation to be specified as an additional parameter and that the dimensionality of the whole storage increases by one.

## 5.4  Composition Change

If a set of actors is not constant over time because actors join or leave, it has to be specified in the *Data* object. The following methods of the *Data*

16

class should be used for this purpose:

- void *active*(const *ActorSet* * *pActorSet*,
    int *actor*,
    int *observation*,
    bool *flag*) – specifies if the given actor of the given set of actors is active at the given observation.

- void *addJoiningEvent*(int *period*,
    const *ActorSet* * *pActorSet*,
    int *actor*,
    double *time*) – stores the time of a period when an actor joins a certain actor set.

- void *addLeavingEvent*(int *period*,
    const *ActorSet* * *pActorSet*,
    int *actor*,
    double *time*) – stores the time of a period when an actor leaves a certain actor set.

The time values in the above methods have to lie in the range $[0, 1]$ with values 0 and 1 representing the start and end point of the period, respectively. For example, assuming that the length of a period is one year, the following code snippet expresses the fact that and the student $i$ leaves the university 3 months after the second observation and resumes the studies half year after the third observation (remember that the indices of observations start with 0):

```
for (int k = 0; k < observationCount; k++)
{
  if (k == 2)
  {
    pData->active(pStudents, i, k, false);
  }
  else
  {
    // This is not strictly necessary,
    // as the actors are active by default
```

```
    pData->active(pStudents, i, k, true);
  }
}

pData->addLeavingEvent(1, pStudents, i, .25);
pData->addJoiningEvent(2, pStudents, i, .5);
```

The details about every exogenous event are collected in instances of the
*ExogenousEvent* class, which are stored in sets of type *EventSet*, one for each
period. These sets can be accessed with the method *pEventSet*(int *period*).

## 5.5   Accessor Methods

Once a *Data* object has been populated, its sub-objects can be accessed in
two ways. A single object can be looked up by its name, like in the
following examples.

```
const ActorSet * pStudents = pData->pActorSet("students");
OneModeNetworkLongitudinalData * pFriendshipData =
  pData->pOneModeNetworkData("friendship");
```

If some operation is to be performed for all sub-objects
oGenericNetworkEffectf a certain type, one can obtain a reference to the
vector, where the subobjects of this type are stored by the *Data* class. For
example, the values of all constant covariates can be printed as follows:

```
const vector<ConstantCovariate *> & rCovariates =
  pData->rConstantCovariates();

for (unsigned i = 0; i < rCovariates.size(); i++)
{
  ConstantCovariate * pCovariate = rCovariates[i];

  cout << pCovariate->name() << endl;

  for (int j = 0; j < pCovariate->pActorSet()->n(); j++)
  {
```

```
      cout << j << " " << pCovariate->value(j) << endl;
  }
}
```

## 6.  Models

## 6.1  Effects

### 6.1.1  Generic Network Effects

**The Basic Idea.**   The class *GenericNetworkEffect* may be used to specify
an effect for a network variable $x^{(r)}$ if its evaluation and endowment
statistics can be conveniently defined as

$$\sum_{i\neq j} x_{ij}^{(r)} f_{ij}(y)$$

and

$$\sum_{i\neq j} (1 - x_{ij}^{(r)}) x_{ij}^{(r)\text{obs}}(t_m) f_{ij}(y^{\text{obs}}(t_m)),$$

respectivelly. This function $f_{ij}(y)$ as well as the change contribution $\Delta_{kij}^X(y)$
should be specified as instances of the *AlterFunction* class and passed as
parameters when creating the effect.

We illustrate the overall process by defining the density effect
$s_{i1}^{\text{net}}(x) = \sum_j x_{ij}$ as a generic network effect.

1. First, let us define the change contribution function by creating a
   subclass of *AlterFunction*. Since in our case $\Delta_{1ij}^X(y) = 1$, we will name
   the new class accordingly:

   ```
   class ConstantOneFunction : public AlterFunction
   {
     virtual double value(int alter);
   };
   ```

   The method *value* should be overridden to return the value of the
   function for the given alter $j$. The ego $i$ is implicit and can be
   accessed by the method *AlterFunction*::*ego*(). Here is the trivial
   implementation of the *value* method:

19

```
double ConstantOneFunction::value(int alter)
{
  return 1;
}
```

2. Similarly, the tie statistic $f_{ij}(y)$ should be defined. However, since in our case $f_{ij}(y) = \Delta^X_{1ij}(y) = 1$, we can reuse the same class *ConstantOneFunction*.

3. Now that the necessary alter functions are implemented, we are ready to create the generic effect:

```
AlterFunction * pChangeContribution =
  new ConstantOneFunction();
AlterFunctoin * pTieStatistic =
  new ConstantOneFunction();
Effect * pEffect =
  new GenericNetworkEffect(pEffectInfo,
    pChangeContribution,
    pTieStatistic);
```

If the change contribution and the tie statistic functions are the same (as for the density effect), the alternative constructor with a single function is recommended:

```
Effect * pEffect =
  new GenericNetworkEffect(pEffectInfo,
    new ConstantOneFunction());
```

**The Anatomy of Alter Functions.** As indicated above, the key method of the *AlterFunction* class is the method *value* that has to be implemented by all concrete functions. However, there is a couple of other methods that may be necessary to override.

- *initialize*(*pData*, *pState*, *period*, *pCache*) – This method is called before evaluating the function for a specific period of the observed data and a fixed state of the dependent variables. Also, functions may take advantage of the given cache object to speed up the overall processing

time by avoiding repetitive computation of common values, like the
number of two-paths, etc. The following example shows the
initialization method in the class *NetworkAlterFunction*, which is the
base class for all alter functions defined on a network:

```
void NetworkAlterFunction::initialize(const Data * pData,
  State * pState,
  int period,
  Cache * pCache)OneModeNetworkAlterFunction
{
  // Do not forget to initialize the base class
  AlterFunction::initialize(pData,
    pState,
    period,
    pCache);

  // Obtain the right network from the state
  this->lpNetwork = pState->pNetwork(this->name());

  // Store the corresponding network cache object
  this->lpNetworkCache =
    pCache->pNetworkCache(this->lpNetwork);
}
```

- *preprocessEgo*(*ego*) – This method is called before calling the *value*
  method for any specific alters. The base implementation simply stores
  the ego, which can be later accessed by using the *ego* method. The
  derived alter functions may override this method to do some other
  ego-related preprocessing.

**Important Alter Functions.**   There is a whole hierarchy of alter
functions implemented in `src/model/effects/generic`, and one should
review the existing functions before writing his or her own.

- *NetworkAlterFunction* – This is the base class for all functions defined
  on a network such as *InDegreeFunction*, *InStarFunction*, etc. The name of
  the network should be given as an argument on the construction of

the function. The following methods are defined for convenient use in the actual derived functions:

- *Network* * *pNetwork*() – Returns the current network this function is computed for.
- *NetworkCache* * *pNetworkCache*() – Returns the corresponding cache object for efficient calculations.

- *OneModeNetworkAlterFunction* – This is the base class for all functions defined on a one-mode network. It is a subclass of *NetworkAlterFunction* and does nothing more than throwing an exception if the underlying network is not a one-mode network.

- *ProductFunction* – A composite function defined as a product of two other alter functions.

- *DifferenceFunction* – A composite function defined as a difference between two other alter functions.

- *ConditionalFunction* – A composite function returning the value of either of two other alter functions depending on the value of a certain predicate.

**Composite Functions.**    Composite functions like *ProductFunction* and *DifferenceFunction* may be used to define new alter funtions without having to implement new classes. For example, consider the mutuality effect with the network $W$ on the network $X$, which is formally defined as $s_{ik}^{\text{net}}(y) = \sum_j x_{ij} w_{ij} w_{ji}$. The change contribution is $\Delta_{kij}^{X}(y) = w_{ij} w_{ji}$, which is not implemented directly as an alter function. However, there is a function *OutTieFunction* that returns the values $w_{ij}$ of the outgoing ties and an analogous function *InTieFunction* that returns the values $w_{ji}$ of the incoming ties. These two atomic functions can be easily combined in a product function. Assuming that $W$ is a friendship network, the complete code defining the mutuality effect looks like this:

```
pEffect = new GenericNetworkEffect(pEffectInfo,
  new ProductFunction(
    new OutTieFunction("friendship"),
    new InTieFunction("friendship")));
```

**Conditional Functions.** The function *ConditionalFunction* is another type of composite functions, which takes three parameters in its constructor – a predicate and two alter functions. The value of the conditional function depends on the value of the predicate. If the predicate holds for a certain alter $j$, the if-function is evaluated and its value is returned as the value of the conditional function. If the predicated does not hold then the else-function is evaluated. The predicate has to be an object of the *AlterPredicate* class. This class functions very much like *AlterFunction* except that the type of the *value* method is `bool` instead of `double`.