The Sieve of Eratosthenes in `contact`

Antonio Natali

Alma Mater Studiorum – University of Bologna
via Venezia 52, 47023 Cesena, Italy
antonio.natali@unibo.it

Abstract This work presents the usage of the `contact` system with reference to the definition of dynamically (re)configurable distributed software systems, by assuming as a case-study a distributed implementation of the *Sieve of Eratosthenes* algorithm.

The repository *Sieve2013.git* (at `137.204.107.21`) includes the following projects (in increasing order of complexity):

1. *it.unibo.sieve.oo*: A model of the sieve system based on core-objects defined during requirement analysis (see Section 2).

   Artifacts: *Interfaces* `IPrime`, `IFilter` etc.

2. *it.unibo.sieveNoPipe*: The sieve system as a pipe of three (statically configured) subjects (that reuse the core-objects of Section 3) and interact via message-passing.

   Artifacts: *Interfaces* `INatural`, `IPrime extends INatural`, `OO reconfigure` operation

3. *it.unibo.sieve.pipe*: A new version of the the sieve system in which the filter chain is implemented as a pipe of subjects interacting via message-passing (see Section 4).

   Artifacts: a new `reconfigure` operation that calls `updateKBandRun`

4. *it.unibo.sieve.pipe.user*: A new sieve system that includes an *user* that can ask for a prime number and stops the computation (see Section 5).

   Artifacts: operations to check for the presence of messages, hidden within the `onMessage?` construct:

   check message operations
   ```
   boolean checkForMsg( String receiver, String msgId, IPolicy policy  ) and
   boolean checkForSignal(String sender, String msgId, boolean mostRecent, IPolicy policy )
   IMessage checkSignal( String sender, String msgId, boolean mostRecent  )
   ```

5. *it.unibo.sieve.observer*: A sieve system that includes an *observer* of the shared space that stops the computation (see Section 6).

   Artifacts: `Subject passive` and `LindaLike.register( java.util.Observer obj )`

Contents

1 Introductionn

This work presents the usage of the `contact` system with reference to the definition of dynamically (re)configurable distributed software systems. The case-study is a distributed implementation of the algorithm known as *Sieve of Eratosthenes* that finds the prime numbers.

An introduction to the `contact` system can be found in [**?**].

1.1 Overview

The repository *sieve2013.git* (at `137.204.107.21`) includes the following projects (in increasing order of complexity):

1. *it.unibo.sieve.oo*: A model of the sieve system based on core-objects defined during requirement analysis (see Section 2).
2. *it.unibo.sieveNoPipe*: The sieve system as a pipe of three (statically configured) subjects (that reuse the core-objects of Section 3) and interact via message-passing.
3. *it.unibo.sieve.pipe*: A new version of the the sieve system in which the filter chain is implemented as a pipe of subjects interacting via message-passing (see Section 4).
4. *it.unibo.sieve.pipe.user*: A new sieve system that includes an *user* that can ask for a prime number and stops the computation (see Section 5).
5. *it.unibo.sieve.observer*: A sieve system that includes an *observer* of the shared space that stops the computation (see Section 6).

Worthwhile to note:

- Modelling of domain entities as conventional objects (see Section 2)
- Subject classes (see Subsection 3.7))
- Specifications of interactions involving classes (instead of instances) (see Subsection 4.1)
- Dynamic reconfiguration of `contact` systems (see Subsection 4.7)
- Interaction operations with time-out (see Section 5))
- Connection-based interactions in `contact` (TODO)

1.2 The system 'Sieve of Eratosthenes'

The initial system configuration of the *SieveSystem* is made of three elements: *i)* a generator of natural numbers (named `intGen`) starting from `3`; *ii)* a filter (named `filter2`) that eliminates all the natural numbers divisible for `2` and *iii)* a collector of the prime numbers (named `sieve`). There is also an entity (named `user`) interested to the results of the computation.

The behavior of the system can be informally described as follows.

> ***The Sieve of Eratosthenes.*** When `filter2` receives from `intGen` a number not divisible by `2` (i.e. `3`), it sends such a number to the `sieve` that stores it as a prime number. Moreover, `filter2` reconfigures the system by creating a new filter (named `filter3`) that will eliminate from the system all the numbers divisible by `3`. The new filter `filter3` works as a new element of the system inserted between `filter2` and the `sieve`. In this way, the next number that reaches the `sieve` will be `5` and a filter for it (`filter5`) will be inserted between the filter for `3` and the `sieve`.
>
> The user could ask the sieve for the element of a given position in the ordered sequence of prime numbers.

2 An object-based model of the sieve system

In order to better understand how a distributed (heterogeneous) system can be built by (re)using conventional object models defined during the requirement or the problem analysis, let us briefly introduce a pure object oriented (oo) version of the system. The workflow is summarized as follows:

1. REQUIREMENT/PROBLEM ANALYSIS: Model of each domain entity as a "pure" data objects described by an core-interface associated with a proper test-plan.

   core-interfaces
   ```
   public interface IPrime {
           public String getRep();
           public String getProloglikeRep(); //returns int( n )
           public int getAsInt();
   }
   public interface IIntGen {
            public int genNextInt();
   }
   public interface IFilter  {
           public IPrime getMyPrime();
           public boolean handleNextInt(int v); //returns true if 'captured' by the filter
   }
   public interface ISieve {
           public void newPrime( IPrime v );
           public int getNumOfPrimes();
           public IPrime getPrimeAtPos(int i);
   }
   ```

   The test-plan is left to the reader.
2. DESIGN: Model of each domain entity as an object that implements a core-interface and interacts with other objects

   interfaces of the domain entities as objects
   ```
   public interface IEntity { public void doJob();}
   public interface IPrimeObj extends IPrime, IEntity{}
   public interface IIntGenObj extends IIntGen, IEntity{}
   public interface IFilterObj extends IFilter, IEntity{}
   public interface ISieveObj extends ISieve, IEntity{}
   ```

3. DESIGN: (Model of) a system-configuration object

   configuration operations
   ```
           protected void createElements(){
                    sieve = new SieveObj("sieve");
                   filter2 = new FilterObj( "filter2", new PrimeObj("prime2",2),sieve);
                   intGen = new IntGenObj("intGen",filter2);
           }
           protected void start(){ intGen.doJob();        }
   ```

   Each object is configured (i.e. connected with other objects) at creation time.

2.1 OO (domain) model

The behavior of each domain object is defined by applying proper design patterns. For example:

FilterCore and FilterObj.

FilterCore

```
public class FilterCore implements IFilter{
protected IPrime myPrime;
	public FilterCore(IPrime myPrime){ this.myPrime = myPrime;	}
	@Override
	public boolean handleNextInt(int v) { return (v % myPrime.getAsInt())  == 0 ;	 }
	@Override
	public IPrime getMyPrime() { return myPrime;	}
}
```

FilterObj (as a *decorator* [?] of FilterCore)

```
package it.unibo.sieve.oo.impl;
import it.unibo.is.interfaces.IOutputView;
import it.unibo.sieve.oo.core.*;
import it.unibo.sieve.oo.interfaces.*;
public class FilterObj extends Entity implements IFilterObj{
protected IFilter filterCore ;
protected ISieve sieve ;
protected IFilterObj nextFilter ;
	public FilterObj(String name, IPrime prime, IOutputView outView ){
		super(name,outView);
		filterCore = new FilterCore(prime);
	}
	public FilterObj(String name, IPrime prime, ISieve sieve ){
		super(name,null);
		this.sieve	= sieve;
		filterCore = new FilterCore(prime);
	}
	public FilterObj(String name,IPrime prime, IFilterObj filter){
		super(name,null);
		nextFilter	= filter;
		filterCore = new FilterCore(prime);
	}
	@Override
	public boolean handleNextInt(int v) {
	boolean todiscard = filterCore.handleNextInt(v);
		if(  todiscard ) {
			showMsg( "		DISCARDS: "  +  v );
			return true;
		}
		sendIntToNext(v);
		return false;
	}
	@Override
	public IPrime getMyPrime() { return filterCore.getMyPrime(); }
	@Override
	public void doJob(){}
	protected void sendIntToNext(int curVal) {
		if (nextFilter != null) {
			showMsg("		DELEGATES TO NEXT: " + curVal);
			nextFilter.handleNextInt(curVal);
			return;
		}
		// FOUND NEW PRIME
		IPrime vp = new PrimeObj("prime" + curVal, curVal);
		reconfigure(vp);
		sendNewprimeTosieve(vp);
	}
	protected void reconfigure(IPrime vp){
		showMsg("	*** NEW PRIME: " + vp.getRep());
		nextFilter = createAnotherFilter(vp);
	}
	protected IFilterObj createAnotherFilter(IPrime vp) {
		return new FilterObj("filter" + vp.getRep(), vp, sieve);
	}
	protected void sendNewprimeTosieve(IPrime vp) {
		sieve.newPrime(vp);
	}
}
```

3 From the oo sieve system to a first distributed system

Let us define here the sieve system as a pipe of subjects interacting by message-passing. Each subject is conceived as a new decorator of a core object introduced in Section 2. The filter chain is designed and implemented as a chain of objects interacting via method calls, as done in Subsection 2.1. No user subject is introduced; the computation terminates when `intGen` generates the number `0`[1].

3.1 Logic architecture: structure and interaction

The following "code" is a model, expressed in the custom language/metamodel called `contact` [?]. This model defines the *initial configuration* of the application as a system composed of three active entities (the *subjects*) that interact via message-passing.

sieveNoPipeWithObj.contact

```
ContactSystem sieveNoPipeWithObj   ;
Subject class FilterN  ;
Subject intGen   -w;
Subject sieve    -w;
Subject filter2 inherits FilterN    -w;

Dispatch nextInt ;
Invitation newPrime ;

sendIntToFilter2 : intGen forward  nextInt to filter2 ;
recFilter2Int :  filter2 serve nextInt ;

sendPrimeToSieve : FilterN ask newPrime to sieve ;        //CLASS-LEVEL INTERACTION SPEC
recNewprime : sieve accept newPrime ;
```

The subject `filter2` is defined as a specialized version of a class of subjects named `FilterN`. Any object of class `FilterN` is able to *ask* the invitation `newPrime` to the `sieve` subject, while only the subject `filter2` is able to receive the dispatch `nextInt` (sent from the `intGen` subject).

Modelling application message content.
The current version of `contact` (`1.6.12`) does not allow to formally specify the application content of messages [2]. However, in order to interact, two entities must agree on the format of the message exchanged at application level. We[3] state here that:

- The content of the *dispatch* `nextInt` is modelled as a Prolog structured `Term` of the form `natural(n)`.
- The content of the *invitation* `newPrime` is modelled as a Prolog structured `Term` of the form `prime(n)`.

As a consequence, we introduce a proper extension to the domain model of Section 2.

3.2 Re-factoring the models of the domain data

The new model of the domain data is now re-defined by introducing the concept of natural number and the idea that each data-entity must provide one or more operations that build a proper string-based representation of that entity :

---

[1] i.e. the message content `natural(0)`, see Subsection **??**
[2] A proposal to achieve this goal could be studied by the reader.
[3] The reader should evaluate if this specification is introduced in some analysis phase or in a design phase.

New core-interfaces

```
public interface IPrime extends INatural{}

public interface INatural {
        public String getRep(); //a default rep for output purposes
         public int getAsInt();
         public String getProloglikeRep(); //returns natural( n )
}
public interface IIntGen {
         public INatural genNextInt();
}
```

The new test-plan is left to the reader.

For each interface we introduce an implementation class that provides (if it is the case) a set of ***factory methods*** [**?**] in order to facilitate the work of the application designer. Moreover, a ***class invariant*** is introduced, when appropriate.

Natural numbers.

The class Natural

```
public class Natural implements INatural {
        protected int v;

        public static INatural create(int v) throws Exception {
                return new Natural(v);
        }

        public static INatural parsePrologRep(String rep) throws Exception {
        //ASSUMPTION: rep = natural( n ) (as built by getProloglikeRep)
                return new Natural(parsePrologRepToInt(rep));
        }

        public static int parsePrologRepToInt(String rep) throws Exception {
        //rep = natural( n ) (built by getProloglikeRep)
                Struct rt = (Struct) Term.createTerm(rep);
                int val = Integer.parseInt("" + rt.getArg(0));
                return val;
        }

        public Natural(int v) throws Exception {
                invariant(v);
                this.v = v;
        }

        @Override
        public String getRep() {
                return "" + v;
        }

        @Override
        public int getAsInt() {
                return v;
        }

        @Override
        public String getProloglikeRep() {
                return "natural(" + v + ")";
        }

        protected void invariant(int v) throws Exception {
                if (v < 0)
                        throw new Exception("Natural creation error");
        }
}
}
```

Prime numbers.

Prime numbers

```
public class PrimeCore implements IPrime {
	protected int v;

	public static IPrime create(int v) throws Exception {
		return new PrimeCore(v);
	}

	public static IPrime create(String prologRep) throws Exception {
		// ASSUMPTION: prologRep = prime( n ) (as built by getProloglikeRep)
		return new PrimeCore(Natural.parsePrologRepToInt(prologRep));
	}

	public PrimeCore(int v) throws Exception {
		this.v = v;
		invariant(v);
	}

	@Override
	public String getRep() {
		return "" + v;
	}

	@Override
	public int getAsInt() {
		return v;
	}

	@Override
	public String getProloglikeRep() {
		return "prime(" + v + ")";
	}

	protected void invariant(int v) throws Exception {
		if (v < 0)
			throw new Exception("PrimeCore creation error");
		// check that v is divisible for 1 and itself only (TODO)
	}
}
```

FilterCore.

FilterCore

```
public class FilterCore implements IFilter {
	protected IPrime myPrime;

	public static IFilter create(IPrime myPrime) {
		return new FilterCore(myPrime);
	}

	public static IFilter create(int v) throws Exception {
		return new FilterCore(new PrimeCore(v));
	}

	public FilterCore(IPrime myPrime) {
		this.myPrime = myPrime;
	}

	@Override
	public boolean handleNextInt(int v) {
		return (v % myPrime.getAsInt()) == 0;
	}

	@Override
	public IPrime getMyPrime() {
		return myPrime;
	}
}
```

IntGenCore.

IntGenCore

```
public class IntGenCore implements IIntGen {
	protected int n = 3;

	public static IIntGen create() {
		return new IntGenCore();
	}

	@Override
	public INatural genNextInt() {
		try {
			return new Natural(n++);
		} catch (Exception e) {
			return null;
		}
	}
}
```

3.3 The first prototype (integration plan)

While waiting for a proper extension of our custom meta-model related to message contents, we introduce here some comment in the interaction section of the model of Subsection 3.1, in order to recall the structure of the contents of the messages required at application level. We also introduce the specification of *contexts* and network protocols, in order to allow the testing in a true distributed environment.

sieveNoPipeWithObj.contact

```
ContactSystem sieveNoPipeWithObj   ;
Context ctxIntGen -w;
Context ctxFilters -w;
Context ctxSieve -w;
Context ctxTesting  ;

Subject class FilterN  ;
Subject intGen context ctxTesting  -w;
Subject sieve  context ctxTesting  -w;
Subject filter2 inherits FilterN  context ctxTesting  -w;

Dispatch nextInt ;
Invitation newPrime ;

sendIntToFilter2 : intGen forward  nextInt to filter2 ; //nextInt content -> natural(n)
recFilter2Int :  filter2 serve nextInt support=TCP [host="localhost" port=8010];

sendPrimeToSieve : FilterN ask newPrime to sieve ;        //newPrime content -> prime(n)
recNewprime : sieve accept newPrime support=TCP [host="localhost" port=8020];
```

The logic architecture defined by this `contact` model can lead to a ***work plan*** in which each context can be designed, developed and tested by a different group of workers that can use this model as their common project (cognitive) map. Before starting the work in parallel, the software development team should complete the specification with the logic behavior for each subject, in order to fully specify the required interaction among the subjects.

3.4 Behavior of the subject intGen

The subject `intGen` is conceived as a new decorator of the object `intGenCore` introduced in Subsection 3.2 that enables such an object to interact with the subject `filter2` by forwarding to it a message of type `nextInt` with content `natural(n)`.

```
                                BehaviorOf intGen
BehaviorOf intGen {
var it.unibo.sieve.oo.interfaces.IIntGen intGenCore = null
var it.unibo.sieve.oo.interfaces.INatural zero = null
var it.unibo.sieve.oo.interfaces.INatural n = null
var nAsInt = 0
var int maxN = 22

 state  initGeninit  initial onException intGenError
        println( " START "        )
        set intGenCore = call it.unibo.sieve.oo.core.IntGenCore.create()
        set zero = call it.unibo.sieve.oo.core.Natural.create( const.0 )
        set n = zero
        goToState genInt
 endstate
 state  genInt
        set n = call intGenCore.genNextInt()
        set nAsInt = call n.getAsInt()
        println( "genInt | n=" + nAsInt  )
        doOut sendIntToFilter2( call n.getProloglikeRep() )
        if{ nAsInt > maxN } { goToState intGenEnd }
        goToState genInt
 endstate
 state intGenEnd
        doOut sendIntToFilter2( call zero.getProloglikeRep() )
        transitToEnd
 endstate
 state intGenError
         println( "Error " + code.curException)
        transitToEnd
 endstate
 }
```

3.5 Behavior of the subject sieve

The subject `sieve` receives an invitation of type `newPrime` with content `prime(n)` and stores the new prime (operation `storePrime`) in a local data structure.

```
                                BehaviorOf sieve
BehaviorOf sieve {
var msg = ""
var it.unibo.sieve.oo.interfaces.IPrime curPrime

 action void storePrime(it.unibo.sieve.oo.interfaces.IPrime vp)

 state initSieve initial
        println( "START "         )
        onMessage newPrime transitTo handleNewPrime
 endstate
 state handleNewPrime onException sieveError
        set msg = code.curInputMsgContent                //msg -> prime(n)
        if{  msg.contains("0") } { goToState sieveEnd }
        set curPrime = call it.unibo.sieve.oo.core.PrimeCore.create(msg)
        println( "received " + msg ) //msg -> prime(n)
        exec storePrime( curPrime )
        onMessage newPrime transitTo handleNewPrime
 endstate
 state sieveEnd
        println( "END")
        transitToEnd
 endstate
 state sieveError
        println( "Error " + code.curException)
        transitToEnd
 endstate
}
```

3.6 Behavior of the subject filter2

The subject `filter2` is conceived as a new decorator [?] of objects of class `FilterCore` introduced in Subsection 2.1 that enables core-objects to interact via message-passing with the subjects `intGen` and `sieve`. `filter2` models the behavior of a subject that: *i)* receives a dispatch of type `nextInt` with content `natural(n)` and then *ii)* does nothing if the recived number is divisible by 2 otherwise *iii)* forwards the message (and the work) to (if this exists) the next filter object; if there is no other filter in the chain, *iv)* it sends the received number as a new prime to the `sieve`, creates a new filter object and reconfigures the system by adding this new filter as its next filter of a object chain.

Most of the behavior of `filter2` is defined in the class `filterN` by a set of inherited operations (see also Subsection 3.7):

- `void handleNextValue(String applMsg) //applMsg = natural(n)`
  Main job of a filter. If the `applMsg` is `natural(0)`, it sets to `true` a boolean variable `stop`; otherwise it checks (by using the `filterCore`) if the number must be discarded and if not it calls *sendIntToNext*.

- `void sendIntToNext(String applMsg) //applMsg = natural(n)`
  If there is a `nextFilter` in the chain, it calls the `handleNextValue` operation of this object. Otherwise (if `not stop`) it reconfigures the system (by calling the `reconfigure` operation) and sends the new prime (or the `natural(0) if stop`) to the `sieve`.
- `void reconfigure( )`
  Reconfigures the system by creating a new prime and a new filter for it (that becomes it `nextFilter` in the object chain).

BehaviorOf filter2

```
BehaviorOf filter2 {
var curValue = -1
 state  initFilter2  initial onException filter2Error
        set myfilterCore = call it.unibo.sieve.oo.core.FilterCore.create( const.2 );
        println( "START " + myName        )
        onMessage nextInt transitTo filter2HandleInt
 endstate
 state  filter2HandleInt onException filter2Error
        println( "received " + code.curInputMsg )        //curInputMsg -> natural(n)
        exec handleNextValue( code.curInputMsgContent )  //inherited : calls a method chain
        set curValue = code.curInt
        if{ curValue == 0 } { goToState filter2End }
        onMessage nextInt transitTo filter2HandleInt
 endstate
 state filter2End
        showMsg("END")
        transitToEnd
 endstate
 state filter2Error
        showMsg("Error " + code.curException  )
        transitToEnd
 endstate
}
```

3.7 Behavior of the subject class FilterN

The subject class `FilterN` defines a set of operations that realize the behavior of each filter in the object chain, i.e.: *i)* receive a dispatch of type `nextInt` with content `natural(n)` and then handle the received value by calling the `handleNextValue` operation. Since the filter chain is made of conventional ovjects, and the message-receive phase has been already described (and implemented) by the specialized subject `filter2` (see Subsection 3.6), the state-machine defined within the `FilterN` specification is never executed.

BehaviorOf FilterN

```
BehaviorOf FilterN {
 var it.unibo.sieve.oo.interfaces.IFilter myfilterCore  = null
 var it.unibo.sieve.oo.interfaces.IFilter newfilterCore   = null
 var it.unibo.contact.sievePipeWithObj.FilterN newfilter = null
 var it.unibo.is.interfaces.IBasicEnvAwt myEnv        = null
 val withGui = true         //CONFIGURATION; if true create a new GUI for each filter
 var myName = ""
 var int curInt = 0
 var String newFilterName = null
 var boolean stop = false
 var it.unibo.sieve.oo.interfaces.IPrime curfNprime = null

 //Used to inject a filter in (re)configuration (see createAnotherFilter)
 operation void setFilterCore( it.unibo.sieve.oo.interfaces.IFilter fcore){
       set myfilterCore = code.fcore
 }

 //Called by filter2 and by any filter in the chain
 operation void handleNextValue(String applMsg){         //applMsg = natural(n)
       set curInt = call it.unibo.sieve.oo.core.Natural.parsePrologRepToInt(  code.applMsg  )
       if{ curInt == 0 }{        //stop the work
               set stop = const.true ;
               exec sendIntToNext( code.applMsg );
               { return }
       }
       if{ myfilterCore.handleNextInt(curInt) } {
               println( "                DISCARDS: "  +  curInt );
               { return }
       }
       exec sendIntToNext( code.applMsg )
 }
 /*
 * DELEGATE THE WORK to the next filter in the chain
 */
 operation void sendIntToNext(String applMsg){                //applMsg = natural(n)
       if{ nextFilter != null }{
               //println( "                DELEGATES TO NEXT: "  +  code.curInt );
               call nextFilter.handleNextValue(code.applMsg);
               if{ stop } {showMsg("END DETECTED") };
               { return }
       }
       if{ ! stop } {
               //NEW PRIME FOUND
               set curfNprime = call it.unibo.sieve.oo.core.PrimeCore.create( code.applMsg );
               exec  reconfigure() ;
               exec sendValToSieve( call curfNprime.getProloglikeRep() );
               {return}
       }
       //stop => simply forward the natural to the sieve
       showMsg("END THE PROCESS")
       exec sendValToSieve( code.applMsg )
 }
 operation void sendValToSieve ( String m ){
           doOutIn sendPrimeToSieve( code.m )
           acquireAckFor newPrime
           showMsg("ack --> " + code.curReply)
 }
 /*
 * RECONFIGURE THE OBJECT FILTER CHAIN
 */
  operation void reconfigure(){
       //INSERT A NEW FILTER IN THE FILTER-OBJECT CHAIN
       exec createAnotherFilter(  curfNprime )
 }
 /*
 * CREATE A NEW FILTER OBJECT
 */
  operation void createAnotherFilter( it.unibo.sieve.oo.interfaces.IPrime prime ){
       set newFilterName = "filter" + curInt
       set nextFilter =  eval new it.unibo.contact.sievePipeWithObj.FilterN( newFilterName )
       call nextFilter.setEnv(code.env)        //calls initGui()
       set newfilterCore = call it.unibo.sieve.oo.core.FilterCore.create( code.prime );
       call nextFilter.setFilterCore( newfilterCore )  //Inject the filter core for the prime
```

```
        println( "CREATED new filter obj for prime=" + curInt + " " + newFilterName )
 }
//-----------------------------------------------------------------
 state  initFilter  initial
        println( "NEVER HERE since all the wirk is done by filter2"          )
        transitToEnd
 endstate
```

3.8 Dynamic creation of objects

The operation *createAnotherFilter* of `FilterN` creates a new instance of the class `FilterN` and then (following the same code pattern of the configuration code in the generated class `SieveSystemMain.java`) it injects in the new instance the value of the associated prime number.

The operation `call newfilter.setEnv(code.env)` injects in the instance the same environment of the instance that creates it (`filter2` in this case). However, the application designer can associate a new environment to the created instance by overloading the `initGui` method:

FilterN.java: create a new GUI

```
public class FilterN extends FilterNSupport{
protected static int port = 8098;
        public FilterN(String name) throws Exception {
                super(name);
          }
        public static FilterN create(String name) throws Exception{
                return new FilterN(name);
         }
        protected void initGui(){
                if( ! withGui ) return ;
            env = new EnvFrame( getName(), this, Color.green, Color.black );
            env.init();
            env.writeOnStatusBar(getName() + " | IntGenSupport working ... ",14);
            view = env.getOutputView();
          }
}
```

4 Towards a (full) distributed sieve system

Let us define here a new version of the the sieve system as a pipe of subjects in which the filter chain is implemented as a pipe of subjects interacting via message-passing, instead of a chain of objects interacting via nethod calls, as done in Subsection 2.1 and Section 3. No user subject is introduced; the computation terminates when `intGen` generates the number `0`.

4.1 Logic architecture: structure and interaction

The following model, expressed in the custom language/metamodel `contact` [?], defines the *initial configuration* of the application as done in Subsection 3.1 with some important extension marked with `(***)`.

sieveNoPipeWithObj.contact

```
ContactSystem sieveNoPipeWithObj   ;
Subject class FilterN  ;
Subject intGen   -w;
Subject sieve    -w;
Subject filter2 inherits FilterN    -w;

Dispatch nextInt ;
Invitation newPrime ;

sendIntToFilter2 : intGen forward  nextInt to filter2;
recFilter2Int :  filter2 serve nextInt ;
                                 //nextInt content->natural(n)
sendIntToFilterN : FilterN forward  nextInt to FilterN;// (1) (***)CLASS-LEVEL INTERACTION SPEC
recFilterNInt : FilterN serve nextInt;//(2) (***)CLASS-LEVEL INTERACTION SPEC

sendPrimeToSieve : FilterN ask newPrime to sieve;//(3)CLASS-LEVEL INTERACTION SPEC
recNewprime : sieve accept newPrime support=TCP [host="localhost" port=8020];
                                 //newPrime content->prime(n)
```

Now the model states that any object of class `FilterN` is able:

– `(1)` to forward the dispatch `nextInt` to another (dynamically created) filter;
– `(2)` to serve the dispatch `nextInt`, sent by `intGen` or another, (dynamically created) filter;
– `(3)` (as in Subsection 3.1): to *ask* the invitation `newPrime` to the `sieve` subject;

Note that, at point `(1)`, no concrete destination is specified, but only the fact that it must belong to the `FilterN` class. This means that the `Contact-ide` must generate a send support operation (whose signature is in this case
`hl_FilterN_forward_nextInt_FilterN( String M, String destName )`
that includes an argument (`destName`) to be set by the application designer with the name of the concrete destination subject (see the operation `sendIntToNext` in Subsection 4.6).

Moreover, at point `(2)`, no protocol support is specified: in fact such a specification has no-sense (although technically possible) at class level, since it is meaningful only at subject-instance level. This means that some extra informatio must be given by the application designer when a new subject is created (see the operation `updateKBAndRun` in Subsection 4.6).

4.2 The first prototype (integration plan)

The work-plan of Subsection 3.3 is still valid here, with the difference that the group of workers devoted to the design and development of `filter2` and `FilterN` must perform a more complex job than before.

4.3 Behavior of the subject intGen

The behavior model of the subject `intGen` does not change with respect to Subsection 3.4.

4.4 Behavior of the subject sieve

The behavior model of the subject `sieve` does not change with respect to Subsection 3.4.

4.5 Behavior of the subject filter2

The behavior model of the subject `filter2` is now different from that of Subsection 3.4, since `filter2` now works like any other instance of `FilterN`. Thus the model now states that `filter2` simply initializes its core and then inherits its behavior from the state machine defined by `FilterN`.

BehaviorOf filter2

```
BehaviorOf filter2 {
 state  initFilter2  initial onException filter2Error
        set myfilterCore = call it.unibo.sieve.oo.core.FilterCore.create( const.2 );
        println( "START " + myName        )
        call inheritedBehavior()
        goToState filter2End
 endstate
 state filter2End
        showMsg("END")
        transitToEnd
 endstate
 state filter2Error
        showMsg("Error " + code.curException  )
        transitToEnd
 endstate
}
```

4.6 Behavior of the subject class FilterN

As before (see Subsection 3.7) the subject class `FilterN` defines a set of operations that realize the behavior of each filter, with the difference that now we have to design and build a subject pipe in which subjects interact via the `nextInt` dispatch. Thanks to the software organization introduced in Subsection 3.7 the modification of the behavior model can be limited to the *reconfigure* operation.

However, differently from Subsection 3.7, the class `FilterN` must now specify the state machine that defines the behavior of all the (statically and dynamically created) filter subjects.

The FilterN state-machine

```
BehaviorOf FilterN {
...
//(***) RECEIVE AND HANDLE LOOP
 state  initFilter  initial onException filterNError
        showMsg( "START " + myName        )
        onMessage nextInt transitTo filterNHandleInt
 endstate
 state  filterNHandleInt onException filterNError
        showMsg( "received " + code.curInputMsg )        //curInputMsg -> natural(n)
        exec handleNextValue( code.curInputMsgContent )
        if{ curInt == 0 } { goToState filterNEnd }
        onMessage nextInt transitTo filterNHandleInt
 endstate
 state  filterNEnd
        showMsg( "END "  )
        transitToEnd
 endstate
 state  filterNError
        showMsg( "Error " + code.curException  )
        transitToEnd
 endstate
```

4.7 Dynamic reconfiguration of `contact` systems

In order to dynamically extend a (working) `contact` system with other subjects, we have to execute at application level operations that are automatically generated by the `Contact-ide` within the system-configuration class:

1. Create a new subject.
2. Update the run time knowledge base of the system with information useful to implement the communications patterns of that subject.
3. Start the new subject.

In our case, the *createAnotherFilter* operation of Subsection 3.7 can be reused to perform the step `1`, while the steps `2` and `3` are delegated by a new operation called *updateKBAndRun*. Thus, the reconfiguration of the filter pipe can now be expressed as follows:

The new recoinfigure oiperation

```
BehaviorOf FilterN {
...
 action int getPortNum()

 operation void reconfigure(){
        //INSERT A NEW FILTER IN THE FILTER-SUBJECT CHAIN
        exec createAnotherFilter(  curfNprime )
        exec updateKBAndRun(   )                // (***) UPDATE THE KB and RUN THE NEW FILTER
 }
 operation void updateKBAndRun(   ){
        //UPDATE THE Run-Time Knowledge Base
        set nport =  exec getPortNum()
        println( "****************** reconfigureAndRun *********************** "  + nport  )
        call RunTimeKb.addSubject("TCP" , newFilterName , "nextInt", "localhost", nport );
        //RUN THE SUBJECT
         call nextFilter.start()
 }
```

The action `getPortNum` (to be defined by the application designer) returns an integer used as input-port of the new subject for the `nextInt` dispatch. The class `RunTimeKb` (generated by the `Contact-ide`) is the run time knowledge base of the system; thus it is properly updated by calling the *addSubject* (static) operation.

The getPortNum operation can be defined so to return a different port number at each call. However, since all the generated subjects share here the same virtual machine, the returned value can also be the same for all the filters.

4.8 Filter pool

If we want to associate to each new dynamically created filter a different computational node, we could introduce the idea of filter-pool managed by a *filter-coordinator*. When a new filter is required, a request can be sent to the filter-coordinator that selects a non-working filter-resource and transmits as answer the name and the `IP`-port address of this resource. Moreover, the filter-coordinator could also accept requests to the nodes of the network that intend to provide a new filter resource.

The specification (and implementation) of this kind of behavior is left to the reader.

5 The sieve system with an user

Let us introduce here a new sieve system that includes an *user* in which:

1. the user can ask the `sieve` for the element of a given position in the ordered sequence of prime numbers;
2. if no answer is received from `sieve` within an interval of time `DT` (e.g `DT=1sec`) , the user emit a *stop* signal;
3. each component of the system terminates its activity as soon as the `stop` signal is emitted.

The logical architecture of this new system can be expressed in `contact` as follows:

sieveSystem.contact: the components

```
ContactSystem sieveSystem;
//1 - system components
Subject class FilterN ;

Subject intGen ;
Subject sieve ;
Subject filter2 inherits FilterN ;
Subject user ;

//2 - Message types
Dispatch nextInt ;
Invitation newPrime ;
Request getPrimeNum ;
Signal stop ;

//3 - Communication patterns
sendIntToFilter2 : intGen forward  nextInt to filter2;
intGenSense : intGen sense stop ;

recFilter2Int :  filter2 serve nextInt ;
sendIntToFilterN : FilterN forward  nextInt to FilterN;
recFilterNInt : FilterN serve nextInt;
sendPrimeToSieve : FilterN ask newPrime to sieve;
filterNsense :  FilterN sense stop ;

recNewprime : sieve accept newPrime support=TCP [host="localhost" port=8020];
sieveSense: sieve sense stop ;
sieveGrant : sieve grant getPrimeNum ;

userDemand : user demand getPrimeNum to sieve ;
userEmit : user emit  stop  ;

//4 - Subject behavior specification
```

Modelling application message content.
We suppose here that the application content of each message type is a data structure that can be represented as a `Prolog` term as follows (`N` denotes the representation of a natural number, `N>2`).

Message	Content	Meaning
Invitation `newPrime`	`prime(N)`	`N` is a prime number
Dispatch `nextInt`	`natural(n)`	`N` is a natural number
Request `getPrimeNum`	`getPrime(P)`	get the `P-th` prime
Signal `stop`	`stop()`	terminate the process
Answer	`getPrime(P,N)`	answer to `getPrime(P)`

5.1 Workplan

- Define the application type `RequestForAPrime`.
- Define a prototype by completing the specification of Section 5 with the new behavior of each subject.

6 The sieve with an observer

TODO (At the moment read the code)

7 Reliability of Local Message Sends

From http://doc.akka.io/docs/akka/current/general/message-delivery-reliability.html.

The Akka test suite relies on not losing messages in the local context (and for non-error condition tests also for remote deployment), meaning that we actually do apply the best effort to keep our tests stable. A local tell operation can however fail for the same reasons as a normal method call can on the JVM:

StackOverflowError OutOfMemoryError other VirtualMachineError In addition, local sends can fail in Akka-specific ways:

if the mailbox does not accept the message (e.g. full BoundedMailbox) if the receiving actor fails while processing the message or is already terminated While the first is clearly a matter of configuration the second deserves some thought: the sender of a message does not get feedback if there was an exception while processing, that notification goes to the supervisor instead. This is in general not distinguishable from a lost message for an outside observer.