Entità e comunicazione.

Un framework piccolo piccolo per parlare di comunicazione.

Introduzione.

La prendo alla lontana. Molto alla lontana. All'inizio potrà sembrare strano e, probabilmente, diventerà assolutamente bizzarro all'entrata in scena delle prime porzioni di codice, ma queste poche pagine trattano del significato delle parole “contratto”, “messaggio” e “oggetto”. Per la verità “oggetto” è solo sfiorato. In merito ho idee bizzarre e tendo ad evitare il rischio di contaminare qualcuno con le mie corbellerie. Parlare di messaggi e contratti significa parlare di comunicazione. La corrente e un po' stantia metafora del messaggio inviato all'oggetto è un piccolo pezzo del quadro generale. Emblematico ma piccolo. In senso pieno un oggetto riceve qualcosa che contiene anche un messaggio e quel messaggio è parte di una comunicazione che solo se intesa nel suo pieno senso è in grado di esprimere efficacemente il “perchè” del funzionamento di un sistema orientato agli oggetti. Dico perchè intendendo l'insieme di condizioni, implicite o esplicite, che consentono di spiegare l'effettività della metafora del messaggio.
Parlare di comunicazione è necessariamente parlare di significati. In particolare di significati convenzionali, mutabili e immutabili. L'immagine da cui parto, la mia “metafora”, tratta di questo, di significati e di persistenza del significato. Da qui alla comunicazione, il passo è breve. Tratta anche di autonomia e indipendenza degli oggetti. Uso un'autonomia esplicita e, potrei dire, estrema. Mi servo di questa particolare rappresentazione perchè sia del tutto evidente, nel sistema, la parte relativa alla comunicazione. Anticipo, per chiarire. Se costringo ogni oggetto a non poter sapere nulla di nessuno a parte il fatto che tutti possono essere destinatari di una comunicazione dovrebbe risaltare il fatto che ogni interazione è il frutto dell'invio di un qualche significato a qualche destinatario. Perchè la cosa funzioni, è sufficiente definire una piattaforma per lo scambio di messaggi sufficientemente generica da permettere a chiunque di dire qualsiasi cosa. Il framework proposto supporta un tipo di comunicazione asincrona e unidirezionale: un oggetto può solo ricevere e non ha garanzie d'ordine circa la successione dei messaggi che riceve. Infine, i meccanismi proposto hanno uno scopo meramente esemplificativo. Non che la realtà sia molto più complicata. Essa richiede tuttavia alcuni piccoli accorgimenti che, ai fini di questo scritto, ho complessivamente considerato fuorvianti.

I termini di una metafora.

Giro la testa e osservo la stanza in cui mi trovo. Vedo una libreria ed un bel po' di volumi. Ne prendo uno, lo nascondo e torno ad osservare la libreria. Per quanto possa apparire incredibile, la libreria non si è disintegrata in un mare di segatura né i libri rimasti sono diventate orrende copie informi dello splendore che erano.
Trascurando per un attimo l'incipiente follia di cui queste elucubrazioni sono il chiaro segno, ecco il punto: la persistenza del significato di un fenomeno è indifferente all'integrità delle parti di cui è composto.
L'affermazione andrebbe presa cum grano salis. E' più che probabile che dei costituenti di un fenomeno alcuni siano necessari al suo significato, altri lo influenzino e altri ancora siano indifferenti. Il problema, classico e irrisolto, è la definizione di un criterio di essenzialità.
Non ho scelto a caso la libreria. Praticamente ho barato. Posso rimuovere tutti i libri e comunque ricondurre il fenomeno percepito all'idea di libreria, qualcosa che può contenere dei libri anche se attualmente non li contenga.

Posso scrivere un software come se fosse quella libreria?
Scrivo il codice sorgente del programma dividendolo in tanti moduli. Ottengo il programma dalla compilazione di tutti questi pezzi. Dopodichè prendo un pezzo a caso, lo elimino e ciònonostante posso ancora compilare ed eseguire con successo il programma.

Esistenza e interazione.

Dopo un po' di arrovellamenti ho concluso che un meccanismo generale che permetta un certo grado di indifferenza del sistema alla persistenza dei suoi componenti richiede non più di due elementi. Il primo elemento è la comunicazione. La comunicazione è uno scambio di informazioni. In grammatica, la comunicazione è composta di sette elementi:

```
mittente
ricevente
contesto
lingua
canale
messaggio
referente
```

Diciamo che il fenomeno simulato nel programma rappresenta un contesto univoco. Ne restano sei: qualcuno dice a qualcun'altro qualcosa in riferimento a qualcos'altro, il tutto espresso in una certa lingua. Per ora manca il canale: il medium attraverso cui la comunicazione passa dal mittente a ricevente.

Supponendo che la lingua sia simbolica e che in essa siano contenute le rappresentazioni delle diverse identità dei mittenti e dei riceventi, possiamo dichiarare un atto di comunicazione in quesi termini:

```
public class Comm {
   private int to, from, message;
   private Object referent;
}
```

Vale a dire che l'identità del mittente, l'identità del ricevente ed il messaggio sono simboli definiti nel linguaggio, tanto quanto lo sono le parole “mario”, “giuseppe” e “riempilo di mele” e il referente è una cosa qualsiasi – ad esempio un sacchetto vuoto.

Affinchè lo scambio di un messaggio produca un qualche effetto è necessario avere almeno un destinatario potenzialmente attivo: qualcuno che “comprenda” il messaggio e, se del caso, generi le opportune conseguenze. Chiamo questo soggetto Entità e la definisco:

```
public interface Entity{
   int getId();

   void receive(Comm c);
}
```

La proprietà “id” è il segnale che identifica un certo soggetto – o una certa categoria di soggetti. Non corrisponde pienamente al concetto di identità dell'orientamento agli oggetti. Più che un carattere necessario dell'ente, id è un discriminante della comunicazione. Se dico qualcosa in una stanza dove ci sono cento persone, tutte e cento sono in grado di sentire ciò che dico e di reagire come pensino sia più opportuno. Il fatto che un messaggio abbia un certo destinatario rende la comunicazione potenzialmente selettiva. La selezione è fondata sulla possibilità degli ascoltatori di identificare sé stessi come destinatari del messaggio. Il possesso di un'identità o, meglio, la coscienza della propria identità, è uno

degli elementi che determinano il significato di un messaggio. Il metodo getId() rappresenta esattamente questa “coscienza dell'identità” che ogni entità ha riguardo a sé stessa.

Entity e Comm sono i costituenti espliciti del minimondo in costruzione. Considero il resto variabile.

Elementi di base.

Questa è la versione concreta di Entity e Comm che uso nel testo.

```
package it.tukano.ec;

/** Rappresentazione di un'entità */
public interface Entity {

   /** Restituisce l'identità di quest'entità */
   int getId();

   /** L'entità riceve una comunicazione */
   void receive(Comm c);
}
```

```
package it.tukano.ec;

/** Rappresentazione di una comunicazione. La comunicazione è
considerata come lo scambio di un messaggio riferito ad un certo
oggetto tra un mittente ed un destinatario */
public class Comm {
   /* Mittente, destinatario e messaggio */
   private int from, to, message;
   /* Referente della comunicazione. */
   private Object referent;

   /** Inizializza un Comm
   @param from il simbolo che identifica il mittente della comunicazione
   @param to il simbolo che identifica il destinatario della comunicazione
   @param message il simbolo del messaggio scambiato
   @param referent il referente della comunicazione (ciò di cui si parla) */
   public Comm(int from, int to, int message, Object referent) {
      this.from = from;
      this.to = to;
      this.message = message;
      this.referent = referent;
   }

   /** Restituisce l'identità del destinatario del messaggio */
   public int getFrom() {
      return from;
   }

   /** true se la comunicazione abbia come mittente l'entità di identità from */
   public boolean hasFrom(int from) {
      return this.from == from;
   }

   /** true se la comunicazione abbia come destinatario l'entità di identità from */
   public boolean hasTo(int to) {
      return this.to == to;
   }

   /** true se la comunicazione abbia come messaggio il simbolo in argomento */
   public boolean hasMessage(int message) {
      return this.message == message;
   }

   /** Restituisce una conversione del referente a T se il tipo del referente sia
   compatibile in assegnamento con Class<T> e il referente sia diverso da null.
   Restituisce null negli altri casi */
   public <T> T getReferent(Class<T> c) {
      return
         referent != null && c.isAssignableFrom(referent.getClass()) ?
         c.cast(referent) :
```

```
         null;
      }
   }
```

Ovviamente da soli non combinano nulla ma, come anticipato, se escludiamo queste due definizioni il resto può essere considerato ad libitum.

Lingua.

La comunicazione richiede che i comunicanti abbiano almeno una lingua in comune. Definisco la lingua con un'interfaccia.

```
package it.tukano.ec.basic;

public interface BasicLanguage {

   int CHANNEL = 1;
   int SET_CHANNEL = 2;
   int ANYONE = 3;
   int ADD_ENTITY = 4;
   int REMOVE_ENTITY = 5;
}
```

Uso dei valori numerici costanti e condivisi perchè mi sembra che esprima meglio tanto la convenzionalità quanto la simbolicità del linguaggio. I valori contenuti in BasicLanguage sono un insieme minimo necessario ad una certa versione del meccanismo – una delle molte possibili e probabilmente la più elementare. I valori vanno considerati per quello che suggeriscono: sono né più né meno che simboli il cui significato è presupposto essere noto ad ogni entità che li usi.

Entità di base.

Per mia comodità definisco alcune capacità di base per ogni entità. Tali capacità sono destinate alla sola entità la quale, esternamente, non potrà che essere “manipolata” per il suo semplice essere Entity.

```
package it.tukano.ec.basic;

import static it.tukano.ec.basic.BasicLanguage.*;
import it.tukano.ec.*;

/** Definizione di alcune capacità utili ad un'entità di base */
public abstract class BasicEntity implements Entity {
   /* simbolo dell'identità */
   private final int id;
   /* entità canale */
   private Entity channel;

   /** Inizializza un'entità con l'identità in argomento */
   public BasicEntity(int id) {
      this.id = id;
   }

   /** @inheritDoc */
   public int getId() {
      return id;
   }

   /** Qualora la comunicazione sia destinata ad questa
   entità ed il messaggio sia BasicLanguage.SET_CHANNEL, il referente
   della comunicazione è impostato come canale di comunicazione
   per questa entità. Le comunicazioni destinate a questa entità
   sono rinviate per la gestione al metodo processComm. Le comunicazioni
   non destinate a questa entità sono rinviate per la gestione
   al metodo catchComm */
   public void receive(Comm c) {
      if(c.hasTo(getId())) {
         if(c.hasMessage(SET_CHANNEL)) {
```

```
            channel = c.getReferent(Entity.class);
         }
         processComm(c);
      } else {
         catchComm(c);
      }
   }

   /** L'entità gestisce una comunicazione non destinata a sè */
   public abstract void catchComm(Comm c);

   /** L'entità gestisce una comunicazione a sè destinata */
   public abstract void processComm(Comm c);

   /** L'entità invia una comunicazione attraverso il canale, se
   presente */
   public void sendComm(Comm c) {
      if(channel != null) {
         channel.receive(c);
      }
   }

   /** Restituisce il canale di comunicazione usato da questa
   entità */
   public Entity getChannel() {
      return channel;
   }
}
```

Dovrebbe notarsi come l'entità di base richieda anche un simbolo di base per qualificare un messaggio predefinito (SET_CHANNEL).

L'entità canale.

In una comunicazione *vis-a-vi*, i messaggi scambiati tra due persone sono trasportati dall'aria. L'aria è il canale che consente a Tizio di inviare un messaggio e a Caio di riceverlo. Le condizioni affinchè possa esserci scambio di messaggi sono due: che esista un canale e che tale canale sia direttamente o indirettamente condiviso tra mittente e ricevente. Io rappresente il canale come un'entità che mantiene un registro delle entità che lo condividono. Come ogni entità di base un canale gestisce due categorie di comunicazioni: quelle destinati al canale e quelle non destinati al canale. Le comunicazione destinate al canale sono “comprese” se abbiano come messaggio ADD_ENTITY o REMOVE_ENTITY. Il primo messaggio è interpretato come una richiesta di collegamento di un'entità. Il secondo è una richiesta di disconnessione. Ogni comunicazione, sia o non sia destinata al canale, è propagata a tutte le entità collegate al canale. Infine, il canale accumula le comunicazioni destinate ad una certa entità se questa non sia stata connessa. Un po' come se le parole restassero sospese a mezz'aria nell'attesa che il destinatario entri in gioco. E' una palese ma comoda finzione.

```
package it.tukano.ec.basic;

import it.tukano.ec.*;
import static it.tukano.ec.basic.BasicLanguage.*;
import java.util.*;

/** Entità canale di base. Il canale mette in comunicazione più entità
consentendo loro di scambiarsi dei messaggi */
public class BasicChannel extends BasicEntity {
   private ArrayList<Entity> entities = new ArrayList<Entity>();
   private ArrayList<Comm> waitingComms = new ArrayList<Comm>();

   /** Inizializza questa entità usando il valore BasicLanguage.CHANNEL
   come identità */
   public BasicChannel() {
      super(CHANNEL);
   }

   /** Propaga le comunicazioni a tutte le entità connesse. Accumula
   le comunicazione non destinate ad ANYONE o a un'entità già presente. Le
   comunicazione accumulate sono trasmesse all'entità destinataria quando
```

```
   (e se) questa entri in gioco */
   public void catchComm(Comm c) {
      boolean found = false;
      for(Entity e : entities) {
         e.receive(c);
         /*
         1. non ha ancora trovato il destinatario
         2. il messaggio non è per chiunque
         3. il messaggio non è destinato a questa entità
         4. il messaggio è destinato all'entità "e"
         */
         if(!found && !c.hasTo(ANYONE) && !c.hasTo(getId()) && c.hasTo(e.getId())) {
            found = true;
         }
      }
      if(!found) {
         waitingComms.add(c);
      }
   }

   /** Esamina le richieste destinate al canale (ADD_ENTITY e REMOVE_ENTITY) */
   public void processComm(Comm c) {
      if(c.hasMessage(ADD_ENTITY)) {
         Entity e = c.getReferent(Entity.class);
         if(e != null) {
            entities.add(e);
            Comm bind = new Comm(getId(), e.getId(), SET_CHANNEL, this);
            e.receive(bind);
            consumeWaitingComms(e);
         }
      } else if(c.hasMessage(REMOVE_ENTITY)) {
         Entity e = c.getReferent(Entity.class);
         if(e != null) {
            entities.remove(e);
            Comm unbind = new Comm(getId(), e.getId(), SET_CHANNEL, null);
            e.receive(unbind);
         }
      }
      catchComm(c);
   }

   /** Consuma l'elenco di "comunicazioni in attesa del destinatario". Sono
   rimosse dall'elenco quelle comunicazione in attesa destinate all'entità e.
   Le comunicazioni rimosse sono inviate direttamente ad e. */
   private void consumeWaitingComms(Entity e) {
      ArrayList<Comm> sent = new ArrayList<Comm>();
      for(Comm c : waitingComms) {
         if(c.hasTo(e.getId())) {
            sent.add(c);
            e.receive(c);
         }
      }
      waitingComms.removeAll(sent);
   }
}
```

Per inviare una comunicazione attraverso il canale è sufficiente passare l'oggetto Comm al metodo receive di BasicChannel. Sicuramente l'entità canale propagherà la comunicazione ricevuta a tutte le entità collegate. Definirei perciò il modello di comunicazione “aperto”: tutti ascoltano tutto e tutti possono reagire a tutto sebbene idealmente solo alcuni siano in grado di farlo con cognizione di causa. Ad esempio, tutte le entità già collegate al canale riceveranno una comunicazione hasId(CHANNEL) == true, cioè destinata al canale, quando qualcuno voglia dire al canale “collega l'entità X”. Certamente l'entità canale, destinataria esplicita della comunicazione, gestirà quel messaggio. Nulla vieta che altre entità già presenti ragiscano a loro volta in qualche modo. Comm, Entity, BasicLanguage, BasicEntity e BasicChannel esauriscono il necessarie per un pratico uso del modello esposto. Pratico perchè in teoria Comm ed Entity sono gli unici immancabili. A conti fatti, l'insieme dei Basic è una delle possibili forme dell'interazione tra entità comunicanti. C'è solo da notare che sarebbe stato preferibile realizzare BasicChannel come una coda di propagazione di messaggi con annesso Thread propagante. Per volontà sintetica si è omessa la questione ma la realizzazione sarebbe

comunque elementare.

Applicazione.

L'uso del modello proposto pone da subito problemi assolutamente affascinanti. Vi ricordate la libreria da cui sono partito. Ebbene, la mia libreria è necessariamente composta di un Evolution. Evolution è un'entità il cui scopo è generare e, potenzialmente, rigenerare, altre entità. BasicChannel è il canale usato dalle entità per la comunicazione. Avrei potuto fondere i compiti di BasicChannel ed Evolution in una sola entità, chiamiamola Environment, ma comunque avrei poi dovuto ammettere l'esistenza di un problema evolutivo. Il problema è questo: comunque la si metta è sempre necessaria una definizione esterna ad una entità al fine di ottenere l'attivazione del sistema. A seconda di come la vediate o è l'intervento divino o è la casuale aggregrazione di amminoacidi nel brodo primordiale. Per rispetto delle tradizioni, includo una classe Main a sé stante anziché incorporarla in Evolution. La classe Main è, banalmente:

```
package sample;

public class Main {
   public static void main(String[] args) {
      new Evolution();
   }

}
```

Concluso il discorso del main, il sistema richiede un linguaggio più ricco di quanto predisposo in BasicLanguage. Seguendo le orme là tracciate, uso un'altra interfaccia per stabilire ciò che è comunemente noto ad ogni entità del sistema in realizzazione.

```
package sample;

public interface Language extends it.tukano.ec.basic.BasicLanguage {

   int EVOLUTION = 1000;

}
```

Ci sono un paio di cose da dire circa il linguaggio. La prima è che esso è necessariamente condiviso e noto ad ogni entità: è dunque una parte ineliminabile del sistema. Senza linguaggio non c'è possibilità di “compilare” alcunchè. Un po' come le decine di classi delle librerie standard allegramente usate in tutta l'applicazione. La seconda cosa da notare è che tale condivisione necessaria è assolutamente normale. Non c'è alcuna possibilità di interazione tra due elementi se tali elementi non comprendano la stessa lingua, sia essa costituita di parole, voci, gesti o qualsivoglia fenomeno percepibile, così come non c'è possibilità di ricezione se il canale usato per trasmettere non sia in qualche modo disponibile al ricevente. L'interfaccia Language evidenzia questa necessità di condivisione del linguaggio ed ha uno scopo di documentazione della lingua più che di sua effettiva rappresentazione. Detto altrimenti, che il valore EVOLUTION compaia in Language e sia quindi esplicitamente condiviso è indifferente al funzionamento del sistema. Se codificassi nell'entità Evolution 1001 come valore restituito dal metodo getId avrei ottenuto un effetto identico ma avrei mascherato la necessità, per un entità diversa da Evolution, di conoscere un certo simbolo come identificatore di una certa identità. Lo stesso discorso è applicabile alle costanti che segnalano i messaggi. Anziché annotare su un pezzo di carta i simboli compresi da ogni entità li definisco in un'interfaccia e condivido quell'interfaccia. Sarebbe certo preferibile codificare quei valori nelle singole entità o esprimerli attraverso dei riferimenti. Ciò deriva dal particolare trattamento delle costanti (primitivi final) a cui le specifiche del linguaggio di programmazione Java™ acconsentono: è lecito per il

compilatore omettere il riferimento ad un costante sostituendolo con il suo valore. In pratica, se cambia un numero di quelli già assegnati ad un simbolo della lingua occorre ricompilare tutto. La mia preferenza andrebbe alla codifica diretta nelle entità perchè questo sottolinerebbe quella particolare situazione in cui si trova di norma un agente il quale difficilmente “sa tutto ciò che possono dire gli altri” ma è generalmente conscio di un sottoinsieme specifico del linguaggio parlabile. Qui, tuttavia, preferisco evidenziare l'invalcabile necessità della condivisione di un certo insieme di simboli, a prescindere dal fatto che tali simboli siano noti in virtù di una condivisione “diretta” – è il caso dell'interfaccia Language – piuttosto che di una comune conoscenza “indiretta” – ogni entità definisce un linguaggio eventualmente coincidente con la lingua parlata da altre entità.
Vediamo l'entità Evolution:

```
package sample;

import it.tukano.ec.*;
import it.tukano.ec.basic.*;
import static sample.Language.*;

import java.util.*;
import java.io.*;

/** Entità che genera ogni altra entità */
public class Evolution extends BasicEntity {

   /** La creazione di questa entità comporta attivazione
   dell'applicazione */
   public Evolution() {
      super(EVOLUTION);
      ArrayList<Entity> entities = generateEntities();
      if(entities.size() > 0) {
         Entity channel = entities.get(0);
         entities.set(0, this);
         for(Entity e : entities) {
            Comm bind = new Comm(getId(), channel.getId(), ADD_ENTITY, e);
            channel.receive(bind);
         }
      }
   }

   public void catchComm(Comm c) {}

   public void processComm(Comm c) {}

   /** Carica le entità in un array list restituito, a partire da
   un file di testo contenente un elenco dei nomi di queste
   entità */
   private ArrayList<Entity> generateEntities() {
      ArrayList<Entity> entities = new ArrayList<Entity>();
      File file = new File("entities.txt");
      Scanner in = null;
      try {
         in = new Scanner(file);
         ArrayList<String> classNames = new ArrayList<String>();
         while(in.hasNextLine()) {
            classNames.add(in.nextLine());
         }
         loadEntities(classNames, entities);
      } catch(IOException ex) {
         ex.printStackTrace();
      } finally {
         if(in != null) {
            in.close();
         }
      }
      return entities;
   }

   /** Per ogni stringa nell'insieme classNames carica l'Entity corrispondente e
   la accoda ad entities */
   private void loadEntities(Collection<String> classNames, ArrayList<Entity> entities) {
      for(String s : classNames) {
         Entity e = loadEntity(s);
         if(e != null) {
```

```
            entities.add(e);
         }
      }
   }

   /** Caricamento standard della classe className e produzione
   riflessiva di un'istanza */
   private Entity loadEntity(String className) {
      Entity e = null;
      try {
         Class<?> c = Class.forName(className);
         e = (Entity)c.newInstance();
      } catch(ClassNotFoundException ex) {
         System.err.println("Classe non trovata: " + className);
      } catch(IllegalAccessException ex) {
         System.err.println("La classe " + className +
            " non possiede un costruttore vuoto accessibile");
      } catch(InstantiationException ex) {
         System.err.println("Si è verificata un'eccezione " +
            "durante la creazione di un'istanza di " + className);
         ex.printStackTrace(System.err);
      } catch(ClassCastException ex) {
         System.err.println("La classe " + className +
            " non definisce un tipo compatibile con Entity");
      }
      return e;
   }
}
```

A differenza di un'entità comune, qui sta il problema dell'intervento esterno, Evolution non fa qualcosa in risposta ad una comunicazione recepita attraverso il metodo “receive”, normale punto di raccordo tra un'entità e il suo agire. Il suo comportamento iniziale, quello che attiva ogni altra entità, è prodotto direttamente nel costruttore. Avrei potuto stabilire che la stessa cosa fosse fatta in risposta ad una comunicazione? Si ma chi avrebbe generato quella comunicazione? Chi avrebbe detto ad Evolution “carica le entità”? In un Comm di quel tipo necessariemente l'identità del mittente sarebbe stata fittizia – o mitologica. Per sottolineare il problema del giorno zero, anziché scrivere nel main:

```
Evolution evo = new Evolution();
Comm start = new Comm(???, evo.getId(), Language.LOAD_ENTITIES, null);
evo.receive(start);
```

ho scritto direttamente:

```
new Evolution();
```

lasciando che al costruttore il compito di accendere l'universo. Occorre notare che questa è l'unica concessione necessaria ad eventi extra framework. Tutto il resto del sistema è governato dalla ricezione di una comunicazione da parte di un'entità a prescindere dal fatto che esistano altre entità, che una di queste sia un canale o che questo canale colleghi qualcosa. Per semplicità Evolution carica le entità da una lista contenuta in un file di testo. La prima entità della lista è usata come canale. Nulla vieta che tale lista sia altrimenti determinata, ad esempio attraverso una scansione automatica di un percorso prestabilito alla ricerca di classi annotate piuttosto che convenzionalmente nominate o il cui codice byte rechi la dichiarazione di compatibilità in assegnamento con il tipo Entity. Dato Evolution, il sistema è persistente nel senso che risulta attivabile a prescindere dall'esistenza di qualcos'altro. La mancanza di un'altra entità comporta ovviamente un'alterazione più o meno sensibile del risultato. Mancando un'entità che faccia da canale l'applicazione si attiverà e terminerà senza produrre un bel niente. Se introducessi, come farò, una finestra e poi eliminassi la classe in cui tale finestra è contenuta ovviamente il sistema non proietterà più alcuna finestra. Il punto è che il malfunzionamento di una parte non produce il crollo dell'intera applicazione anche se tale malfunzionamento arrivi ad consistere nella privazione di un intero componente dell'applicazione.

Lanciando Evolution tramite il Main, in assenza di un file entities.txt, l'applicazione notifica l'eccezione di I/O correlata alla mancanza del file e poi termina. Introducendo un file entities.txt, vuoto, Evolution non notifica nulla e l'applicazione non fa nulla. Introducendo nel file entities.txt il nome della classe BasicChannel:

```
it.tukano.ec.basic.BasicChannel
```

Evolution si attiva e... non fa un bel niente, cosa che ormai sembra essere diventata la sua specialità. Introduco un'entità che fa poco più di quanto faccia Evolution. L'entità si chiamerà HELLO_WORLD e stamperà un messaggio appena riceva un messaggio di connessione ad un canale. Il primo passo è rendere noto il nome di questa entità. Ribadisco che si tratta di una necessità fittiza e a scopo di documentazione. In Language aggiungo il segno del nome di questa nuova entità.

```
package sample;

public interface Language extends it.tukano.ec.basic.BasicLanguage {

   int EVOLUTION = 1001;
   int HELLO_WORLD = 1002;
}
```

In entities.txt inserisco il nome della classe della nuova entità:

```
it.tukano.ec.basic.BasicChannel
sample.HelloWorld
```

Ora creo l'entità HelloWorld. Iniziamo dalla definizione “vuota”:

```
package sample;

import it.tukano.ec.*;
import it.tukano.ec.basic.*;
import static sample.Language.*;

public class HelloWorld extends BasicEntity {

   public HelloWorld() {
      super(HELLO_WORLD);
   }

   public void processComm(Comm c) {}

   public void catchComm(Comm c) {}
}
```

A questo punto emerge quella necessità di conoscenza dell'insieme di simboli che costituiscono il linguaggio della comunicazione. Voglio stampare un messaggio di testo quando HelloWorld riceva un messaggio che dice “hey, sei stato connesso ad un canale!”. e' una comunicazione destinata ad HELLO_WORLD e da BasicLanguage so che il messaggio è SET_CHANNEL. La “comodità” di estendere BasicEntity sta nel fatto che questo definisce una distribuzione standard delle comunicazioni per cui io so che le comunicazioni destinate a HELLO_WORLD finiscono nel suo metodo “processComm”. Dunque in processComm dirò:

```
if(c.hasMessage(SET_CHANNEL)) {
   System.out.println("Sono VIVO!!! H-E-L-L-O W-O-R-L-D!!!");
}
```

Per esteso:

```
package sample;
```

```
import it.tukano.ec.*;
import it.tukano.ec.basic.*;
import static sample.Language.*;

public class HelloWorld extends BasicEntity {

   public HelloWorld() {
      super(HELLO_WORLD);
   }

   public void processComm(Comm c) {
      if(c.hasMessage(SET_CHANNEL)) {
         System.out.println("Sono VIVO!!! H-E-L-L-O W-O-R-L-D!!!");
      }
   }

   public void catchComm(Comm c) {}
}
```

Compilando Language.java ed HelloWorld.java ed eseguendo, si ottiene il messaggio desiderato sulla console.

A me gli occhi.

Sembra tutto una stupidaggine e magari lo è veramente, ma il modello ha dei curiosi effetti collaterali. Uno è evidente: si può costruire un programma funzionante basandosi su entità che non esistono. Ciò che otteniamo è un sistema che non fa quello che è previsto che faccia ma ciò non impedisce al sistema di esistere nel pieno senso della parola. Prendiamo come entità inesistente un Logger. Supponiamo che anziché stampare la stringa sulla console, HelloWorld la spedisca ad un Logger. HelloWorld non deve sapere se esista e neppure deve sapere quali caratteristiche abbia. Tutto ciò che deve sapere è come parlargli. Stabiliamo che per dire ad un Logger di stampare un testo si debba inviare una comunicazione destinata a qualcuno di nome LOGGER, con messaggio LOG_MESSAGE e referente una stringa di testo. Una cosa del tipo:

```
Comm comm = new Comm(mittente, LOGGER, LOG_MESSAGE, “Hello World!!!”);
```

Fissiamo i nomi nell'interfaccia che documenta il linguaggio:

```
package sample;

public interface Language extends it.tukano.ec.basic.BasicLanguage {

   int EVOLUTION = 1001;

   int HELLO_WORLD = 1002;

   int LOGGER = 1003;

   int LOG_MESSAGE = 1004; //logger, stampami questo!
}
```

Ora cambiamo HelloWorld per farli spedire la stringa al Logger anziché sulla console:

```
package sample;

import it.tukano.ec.*;
import it.tukano.ec.basic.*;
import static sample.Language.*;

public class HelloWorld extends BasicEntity {

   public HelloWorld() {
      super(HELLO_WORLD);
   }

   public void processComm(Comm c) {
```

```
      if(c.hasMessage(SET_CHANNEL)) {
         Comm log = new Comm(
            getId(),
            LOGGER,
            LOG_MESSAGE,
            "Hello World!!!");
         sendComm(log);
      }
   }

   public void catchComm(Comm c) {}
}
```

Compilando Language ed HelloWorld e poi eseguendo l'applicazione non noteremo un bel niente. Possiamo inserire in entities.txt il nome del nostro Logger inesistente e vedere che succede:

```
it.tukano.ec.basic.BasicChannel
sample.HelloWorld
sample.Logger
```

Compare un laconico “classe non trovata: sample.Logger”. Scriviamo il logger. Prima la base:

```
package sample;

import it.tukano.ec.*;
import it.tukano.ec.basic.*;
import static sample.Language.*;

public class Logger extends BasicEntity {

   public Logger() {
      super(LOGGER);
   }

   public void catchComm(Comm c) {

   }

   public void processComm(Comm c) {

   }
}
```

e ora la definizione di una reazione ad un messaggio compreso. Logger sa cosa fare quando qualcuno gli dica “LOG_MESSAGE”. Diciamo che in presenza di questa richiesta produca a video la data di esecuzione e il messaggio proposto:

```
package sample;

import it.tukano.ec.*;
import it.tukano.ec.basic.*;
import static sample.Language.*;

public class Logger extends BasicEntity {

   public Logger() {
      super(LOGGER);
   }

   public void catchComm(Comm c) {

   }

   public void processComm(Comm c) {
      if(c.hasMessage(LOG_MESSAGE)) {
         String text = new java.util.Date() +
            "..." +
            c.getReferent(String.class);
         System.out.println(text);
      }
   }
```

```
}
```

Dopo aver compilato Logger, riavviando l'applicazione noteremo a video il testo che HelloWorld ha prodotto e poi inviato al logger per la stampa.

Un modo complicato per fare cose semplici.

Eliminiamo i file MainWindow.class e il sorgente MainWindow.java. Evolution farà quello che deve ed il programma girerà lo stesso. Vediamo qualche cosa di più pratico. Iniziamo creando un'entità MainWindow che si proietti a video quando riceva una connessione ad un canale. Lavorare con Swing richiede che talune operazioni siano condotte da un Thread dedicato. Onde evitare di scrivere quintali di classi interne definiamo una classe di mero servizio, idealmente destinata ad essere infilata in una libreria “utils” e che, per pigrizia, metto direttamente in sample.

```
package sample;

import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import javax.swing.*;

public class Runfaref extends AbstractAction implements Runnable {
   private Statement statement;

   public Runfaref(String actionName, Object ref, String methodName, Object...args) {
      super(actionName);
      statement = new Statement(ref, methodName, args);
   }

   public Runfaref(Object ref, String methodName, Object...args) {
      statement = new Statement(ref, methodName, args);
   }

   public void actionPerformed(ActionEvent e) {
      run();
   }

   public void invokeLater() {
      SwingUtilities.invokeLater(this);
   }

   public void run() {
      try {
         statement.execute();
      } catch(Exception ex) {
         throw new RuntimeException(ex);
      }
   }
}
```

Con Runfaref, oggetto tuttofare, quando voglio invocare un qualche insieme di istruzione dall'AWT Event Dispatcher Thread dedico a quegli enunciati un metodo pubblico e lo invoco con new Runfaref(this, nomeDelMetodo, argomenti).invokeLater(). Un piccolo esercizio di sintesi e nulla più.
Nome dell'entità: MAIN_WINDOW.

```
package sample;

public interface Language extends it.tukano.ec.basic.BasicLanguage {

   int EVOLUTION = 1001;

   int HELLO_WORLD = 1002;

   int LOGGER = 1003;

   int LOG_MESSAGE = 1004; //logger, stampami questo!
```

```
   int MAIN_WINDOW = 1005;
}
```

Includiamo il nome della classe tra quelle caricate da Evolution:

```
it.tukano.ec.basic.BasicChannel
sample.Logger
sample.MainWindow
```

E, con l'aiuto di Runfaref, definiamo MainWindow senza classi interne, che è sempre un piacere:

```
package sample;

import it.tukano.ec.*;
import it.tukano.ec.basic.*;
import static sample.Language.*;
import javax.swing.*;

public class MainWindow extends BasicEntity {
   private JFrame window = new JFrame();

   public MainWindow() {
      super(MAIN_WINDOW);
      window.setSize(400, 400);
      window.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
   }

   public void catchComm(Comm c) {}

   public void processComm(Comm c) {
      if(c.hasMessage(SET_CHANNEL) && c.getReferent(Entity.class) != null) {
         new Runfaref(this, "showWindow").invokeLater();
      }
   }

   public void showWindow() {
      window.setVisible(true);
   }
}
```

Il metodo showWindow è pubblico per necessità (in verità sarebbe possibile l'invocazione anche se fosse private, per via degli AccessibleObject ma è cosa soggetta a restrizioni di sicurezza). Per noi è indifferente. Niente di ciò che è pubblico in un'entità e diverso da getId o receive è per definizione accessibile senza che ciò violi necessariamente lo scopo prefisso – quello di garantire l'irrilevanza delle mutazioni all'esistena del sistema. Fosse anche un getValue, o lo si invoca con una comunicazione o si butta tutto alle ortiche. E' un prendere o lasciare. L'esecuzione proietterà a video una finestra.
A questo punto facciamo una cosa un po' più strana. Creiamo un'entità che possieda una mappa di stringhe associate a delle chiavi, RESOURCE_MANAGER, e dotiamo di due capacità. ResourceManager comunica a tutti la propria entrata in scena con un messaggio che trasporta la sua mappa di stringhe. ResourceManager è in grado di rispondere ad una richiesta diretta di comunicazione della sua mappa. L'inserimento comporta, al solito, un'evoluzione del linguaggio.

```
package sample;

public interface Language extends it.tukano.ec.basic.BasicLanguage {

   int EVOLUTION = 1001;

   int HELLO_WORLD = 1002;

   int LOGGER = 1003;

   int LOG_MESSAGE = 1004; //logger, stampami questo!

   int MAIN_WINDOW = 1005;
```

```
   int RESOURCE_MANAGER = 1006;

   int TEXT_MAP = 1007;

   /** Una mappa di stringhe */
   class StringMap extends java.util.HashMap<String, String> {};
}
```

e dell'elenco di entità esistenti:

```
it.tukano.ec.basic.BasicChannel
sample.Logger
sample.MainWindow
sample.ResourceManager
```

Una prima versione di quest'entità potrebbe essere:

```
package sample;

import it.tukano.ec.*;
import it.tukano.ec.basic.*;
import static sample.Language.*;

public class ResourceManager extends BasicEntity {
   private StringMap map = new StringMap();

   public ResourceManager() {
      super(RESOURCE_MANAGER);
      map.put("window.title", "Sample - MainWindow");
   }

   public void processComm(Comm c) {
      if(c.hasMessage(SET_CHANNEL) && getChannel() != null) {
         notifyTextMap();
      } else if(c.hasMessage(TEXT_MAP)) {
         sendTextMap(c.getFrom());
      }
   }

   public void catchComm(Comm c) {

   }

   private void notifyTextMap() {
      Comm comm = new Comm(getId(), ANYONE, TEXT_MAP, map);
      sendComm(comm);
   }

   private void sendTextMap(int reqId) {
      Comm comm = new Comm(getId(), reqId, TEXT_MAP, map);
      sendComm(comm);
   }
}
```

Ora l'entità esiste nel sistema ma nessuno è in grado di comprendere ciò che dice. Un possibile interessato c'è. Guarda caso, tra le risorse della mappa di stringhe ce n'è una che si chiama “window.title”. E' possibile far sì che MainWindow acquisisca questo valore e lo proietti come titolo della sua finestra? Certamente si. C'è un messaggio che viaggia nel canale. Come ogni entità, MainWindow lo ascolta. E' sufficiente quindi far si che lo capisca. Basta qualche riga:

```
package sample;

import it.tukano.ec.*;
import it.tukano.ec.basic.*;
import static sample.Language.*;
import javax.swing.*;

public class MainWindow extends BasicEntity {
   private JFrame window = new JFrame();

   public MainWindow() {
```

```
      super(MAIN_WINDOW);
      window.setSize(400, 400);
      window.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
   }

   public void catchComm(Comm c) {
      if(c.hasMessage(TEXT_MAP) && c.hasFrom(RESOURCE_MANAGER) && c.hasTo(ANYONE)) {
         new Runfaref(this, "setLabels", c.getReferent(StringMap.class)).invokeLater();
      }
   }

   public void processComm(Comm c) {
      if(c.hasMessage(SET_CHANNEL) && c.getReferent(Entity.class) != null) {
         new Runfaref(this, "showWindow").invokeLater();
      }
   }

   public void showWindow() {
      window.setVisible(true);
   }

   public void setLabels(StringMap map) {
      window.setTitle(map.get("window.title"));
   }
}
```

La faccenda funziona perfettamente e MainWindow acquisisce il suo titolo. Ma anche no. Per un problema d'ordine. Si modifichi la posizione dei nomi delle classi nel file entities.txt in modo tale che ResourceManager preceda MainWindow:

```
it.tukano.ec.basic.BasicChannel
sample.Logger
sample.ResourceManager
sample.MainWindow
```

Eseguendo il programma si noterà come MainWindow non acquisisca più il titolo della finestra. Il problema è generato dalla comunicazione senza destinatari di ResourceManager (che comunica ad ANYONE) il fatto di essere entrato in scena. BasicChannel accumula le comunicazioni solo se questi siano destinate ad entità diverse da ANYONE: tali comunicazioni, infatti, resterebbero pendenti per tutta la durata dell'applicazione, essendo sempre possibile che una nuova entità si inserisca a sistema avviato. Per ovviare al problema d'ordine possiamo stabilire che MainWindow richieda a ResourceManager la comunicazione delle sue stringhe quando MainWindow entri in gioco.

```
package sample;

import it.tukano.ec.*;
import it.tukano.ec.basic.*;
import static sample.Language.*;
import javax.swing.*;

public class MainWindow extends BasicEntity {
   private JFrame window = new JFrame();

   public MainWindow() {
      super(MAIN_WINDOW);
      window.setSize(400, 400);
      window.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
   }

   public void catchComm(Comm c) {
      if(c.hasMessage(TEXT_MAP) && c.hasFrom(RESOURCE_MANAGER) && c.hasTo(ANYONE)) {
         new Runfaref(this, "setLabels", c.getReferent(StringMap.class)).invokeLater();
      }
   }

   public void processComm(Comm c) {
      if(c.hasMessage(SET_CHANNEL) && c.getReferent(Entity.class) != null) {
         new Runfaref(this, "showWindow").invokeLater();
         requestLabels();
      } else if(c.hasMessage(TEXT_MAP) && c.hasFrom(RESOURCE_MANAGER)) {
```

```
            new Runfaref(this, "setLabels", c.getReferent(StringMap.class)).invokeLater();
         }
      }

      public void showWindow() {
         window.setVisible(true);
      }

      public void requestLabels() {
         Comm req = new Comm(getId(), RESOURCE_MANAGER, TEXT_MAP, null);
         sendComm(req);
      }

      public void setLabels(StringMap map) {
         window.setTitle(map.get("window.title"));
      }
   }
```

Poiché le comunicazioni ad identità univoche sono conservate, nell'attesa che l'entità bersaglio si manifesti, la richiesta delle etichette da parte di MainWindow a ResourceManager è indifferente all'ordine di introduzione delle entità nell'ambiente. Possiamo eliminare da ResourceManager la capacità di notificare le proprie etichette al momento della creazione che è diventata a questo punto rindondante.

```
package sample;

import it.tukano.ec.*;
import it.tukano.ec.basic.*;
import static sample.Language.*;

public class ResourceManager extends BasicEntity {
   private StringMap map = new StringMap();

   public ResourceManager() {
      super(RESOURCE_MANAGER);
      map.put("window.title", "Sample - MainWindow");
   }

   public void processComm(Comm c) {
      if(c.hasMessage(TEXT_MAP)) {
         sendTextMap(c.getFrom());
      }
   }

   public void catchComm(Comm c) {

   }

   private void notifyTextMap() {
      Comm comm = new Comm(getId(), ANYONE, TEXT_MAP, map);
      sendComm(comm);
   }

   private void sendTextMap(int reqId) {
      Comm comm = new Comm(getId(), reqId, TEXT_MAP, map);
      sendComm(comm);
   }
}
```

Tanto per complicare un po' le cose, aggiungo una barra dei menu come entità. Mi servirà per applicare le capacità offerte da una comunicazione “libera” ad un caso un po' più concreto. L'entità si chiamerà MENU_BAR e comunicherà a MAIN_WINDOW l'impostazione di un componente come barra dei menu con un messaggio SET_MENU_BAR. La lingua si amplia:

```
package sample;

public interface Language extends it.tukano.ec.basic.BasicLanguage {

   int EVOLUTION = 1001;

   int HELLO_WORLD = 1002;
```

```
   int LOGGER = 1003;

   int LOG_MESSAGE = 1004; //logger, stampami questo!

   int MAIN_WINDOW = 1005;

   int RESOURCE_MANAGER = 1006;

   int TEXT_MAP = 1007;

   int MENU_BAR = 1008; //Identità della barra dei menu

   int SET_MENU_BAR = 1009; //Messaggio per MAIN_WINDOW

   /** Una mappa di stringhe */
   class StringMap extends java.util.HashMap<String, String> {};
}
```

Quella che segue è una versione momentaneamente incompleta di MenuBar:

```
package sample;

import it.tukano.ec.*;
import it.tukano.ec.basic.*;
import static sample.Language.*;
import javax.swing.*;

public class MenuBar extends BasicEntity {
   private JMenuBar menuBar = new JMenuBar();
   private JMenu file = menuBar.add(new JMenu());
   private JMenu lang = menuBar.add(new JMenu());
   private JMenuItem open = file.add(new JMenuItem());
   private JMenuItem close = file.add(new JMenuItem());
   private JCheckBoxMenuItem ita = (JCheckBoxMenuItem)lang.add(new JCheckBoxMenuItem());
   private JCheckBoxMenuItem eng = (JCheckBoxMenuItem)lang.add(new JCheckBoxMenuItem());

   public MenuBar() {
      super(MENU_BAR);
      ButtonGroup langGroup = new ButtonGroup();
      langGroup.add(ita);
      langGroup.add(eng);
      eng.setSelected(true);
   }

   public void processComm(Comm c) {
      if(c.hasMessage(SET_CHANNEL) && getChannel() != null) {
         requestLabels();
      } else if(c.hasMessage(TEXT_MAP)) {
         StringMap map = c.getReferent(StringMap.class);
         if(map != null) {
            new Runfaref(this, "setLabels", map).invokeLater();
         }
      }
   }

   public void catchComm(Comm c) {

   }

   public void setLabels(StringMap map) {
      file.setText(map.get("menu.file"));
      lang.setText(map.get("menu.language"));
      open.setText(map.get("menu.open"));
      close.setText(map.get("menu.close"));
      ita.setText(map.get("menu.ita"));
      eng.setText(map.get("menu.eng"));
      sendSetMenuBarComm();
   }

   private void sendSetMenuBarComm() {
      Comm c = new Comm(getId(), MAIN_WINDOW, SET_MENU_BAR, menuBar);
      sendComm(c);
   }

   private void requestLabels() {
      Comm c = new Comm(getId(), RESOURCE_MANAGER, TEXT_MAP, null);
      sendComm(c);
   }
}
```

Ciò che MenuBar “dice”, il modo in cui partecipa al sistema, è semplice. Quando riceve una comunicazione che segnala l'acquisito possesso di un canale, considerando dunque questo momento come l'entrata in scena dell'entità MenuBar, MenuBar chiede a qualcuno di nome RESOURCE_MANAGER una TEXT_MAP. Le regole del bon-ton vorrebbero che a tale richiesta qualcuno risponda con un TEXT_MAP il cui contenuto è uno StringMap. Quando MenuBar riceve una comunicazione di cui è il destinatario, con messaggio TEXT_MAP e contenuto di tipo StringMap, considera lo StringMap come contenitore delle etichette per i suoi controlli. Appena MenuBar termina l'assegnazione delle etichette ai pulsanti, dichiara a gran voce “MAIN_WINDOW, SET_MENU_BAR, menuBar”. Potrebbe succedere tutto e niente. MenuBar non sa se riceverà mai una comunicazione SET_CHANNEL né è in grado di determinare se qualcuno possa capire il significato di un RESOURCE_MANAGER, TEXT_MAP e anche se questa comunicazione fosse compresa non è detto che qualcuno possa o voglia rispondere fornendo uno StringMap.
Dopo aver aggiunto MenuBar all'elenco delle entità esistenti:

```
it.tukano.ec.basic.BasicChannel
sample.Logger
sample.ResourceManager
sample.MainWindow
sample.MenuBar
```

possiamo provare ad attivare il mondo di entità per scoprire che nulla è cambiato. Niente appare cambiato perchè nessuno risponde alla chiamata MAIN_WINDOW, SET_MENU_BAR. Insegnamo a MAIN_WINDOW il significato di quella comunicazione.

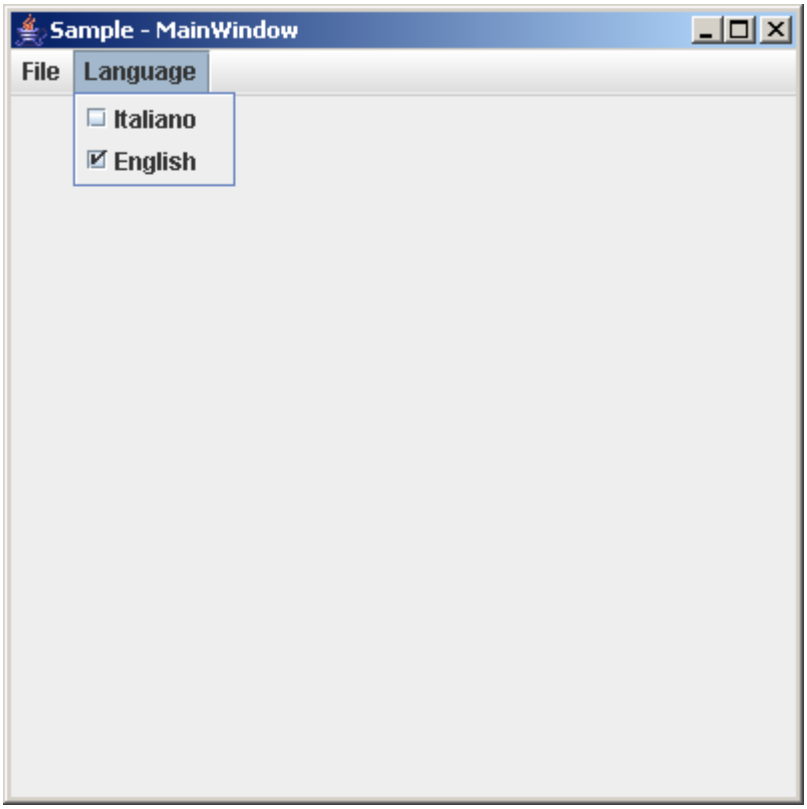
```
package sample;

import it.tukano.ec.*;
import it.tukano.ec.basic.*;
import static sample.Language.*;
import javax.swing.*;

public class MainWindow extends BasicEntity {
   private JFrame window = new JFrame();

   public MainWindow() {
      super(MAIN_WINDOW);
      window.setSize(400, 400);
      window.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
   }

   public void catchComm(Comm c) {
      if(c.hasMessage(TEXT_MAP) && c.hasFrom(RESOURCE_MANAGER) && c.hasTo(ANYONE)) {
         new Runfaref(this, "setLabels", c.getReferent(StringMap.class)).invokeLater();
      }
   }

   public void processComm(Comm c) {
      if(c.hasMessage(SET_CHANNEL) && c.getReferent(Entity.class) != null) {
         new Runfaref(this, "showWindow").invokeLater();
         requestLabels();
      } else if(c.hasMessage(TEXT_MAP) && c.hasFrom(RESOURCE_MANAGER)) {
         new Runfaref(this, "setLabels", c.getReferent(StringMap.class)).invokeLater();
      } else if(c.hasMessage(SET_MENU_BAR)) {
         JMenuBar menuBar = c.getReferent(JMenuBar.class);
         if(menuBar != null) {
            new Runfaref(this, "setMenuBar", menuBar).invokeLater();
         }
      }
   }

   public void setMenuBar(JMenuBar mb) {
      window.setJMenuBar(mb);
      window.validate();
   }

   public void showWindow() {
      window.setVisible(true);
```

```
   }

   public void requestLabels() {
      Comm req = new Comm(getId(), RESOURCE_MANAGER, TEXT_MAP, null);
      sendComm(req);
   }

   public void setLabels(StringMap map) {
      window.setTitle(map.get("window.title"));
   }
}
```

Quando MAIN_WINDOW è destinatario di una comunicazione recante come messaggio SET_MENU_BAR ed un referente di tipo JMenuBar, MAIN_WINDOW assegna quel JMenuBar alla propria finestra. L'esecuzione di questa versione produce una finestra con una barra dei menu appena accennata. L'inghippo sta nella mancanza di etichette per i pulsanti dei menu. MenuBar chiede uno StringMap e, se osserviamo il comportamento di ResourceManager, sappiamo che ResourceManager sa come trattare una richiesta di questo tipo. Sappiamo inoltre che il modo in cui ResourceManager risponde ad una richiesta TEXT_MAP è comprensibile a MenuBar. Il problema è banalissimo: le stringhe che MenuBar vorrebbe assegnare ai suoi controlli non sono contenute nello StringMap di ResourceManager. E' una questione semplice ma è sempre un problema di comunicazione, di condivisione di significati: ciò che MenuBar presuppone che ResourceManager sappia non trova riscontro nella risposta di ResourceManager a MenuBar. I simboli non compresi sono quelli che, per MenuBar, consentono di distinguere le etichette dei controlli: menu.file, menu.language, menu.open, menu.close, menu.ita, menu.eng. Occorre, in un certo senso, insegnare a ResourceManager questi significati affinchè possa comunicarli a MenuBar attraverso uno StringMap.

```
package sample;

import it.tukano.ec.*;
import it.tukano.ec.basic.*;
import static sample.Language.*;

public class ResourceManager extends BasicEntity {
   private StringMap map = new StringMap();

   public ResourceManager() {
      super(RESOURCE_MANAGER);
      map.put("window.title", "Sample - MainWindow");
      map.put("menu.file", "File");
      map.put("menu.language", "Language");
      map.put("menu.open", "Open");
      map.put("menu.close", "Close");
      map.put("menu.ita", "Italiano");
      map.put("menu.eng", "English");
   }

   public void processComm(Comm c) {
      if(c.hasMessage(TEXT_MAP)) {
         sendTextMap(c.getFrom());
      }
   }

   public void catchComm(Comm c) {

   }

   private void notifyTextMap() {
      Comm comm = new Comm(getId(), ANYONE, TEXT_MAP, map);
      sendComm(comm);
   }

   private void sendTextMap(int reqId) {
      Comm comm = new Comm(getId(), reqId, TEXT_MAP, map);
      sendComm(comm);
   }
}
```

L'introduzione della comprensione di questi significati in ResourceManager consente al dialogo tra le entità di produrre il risultato desiderato.
A questo punto attribuiamo un significato alla pressione dei pulsanti del linguaggio.



È ancora un problema di comunicazione: qualcuno dirà qualcosa e qualcun'altro, comprendendo ciò che si dice, risponderà in qualche modo. L'intreccio consapevole della comunicazione tra le entità è ciò che produce un qualche effetto, sperabilmente quello voluto, nel sistema.
Alla pressione di uno dei pulsanti della localizzazione faccio corrispondere l'esclamazione: RESOURCE_MANAGER, SET_LOCALE, String. La stringa corrisponde all'identificatore della località desiderata. Possiamo comprendere sin d'ora chi siano i partecipanti a quest'allegra chiacchierata. MENU_BAR comunica il messaggio che inizia il dialogo. RESOURCE_MANAGER, che possiede le stringhe dei vari testi usati nel sistema, lo comprende e risponde caricando un nuovo insieme di testi, corrispondente alla località richiesta. RESOURCE_MANAGER comunica a chiunque l'avvenuto aggiornamento dei testi. Tutti gli interessati si dimostreranno tali catturando questo messaggio e interpretandolo secondo le proprie esigenze. Gli interessati a nostra disposizione sono due: MAIN_WINDOW, che usa una stringa per il titolo della finestra, e MENU_BAR, che usa le stringhe per i testi dei suoi pulsanti. Tutto questo richiede una nuova "parola": SET_LOCALE.

```
package sample;

public interface Language extends it.tukano.ec.basic.BasicLanguage {

   int EVOLUTION = 1001;

   int HELLO_WORLD = 1002;

   int LOGGER = 1003;

   int LOG_MESSAGE = 1004; //logger, stampami questo!

   int MAIN_WINDOW = 1005;
```

```
   int RESOURCE_MANAGER = 1006;

   int TEXT_MAP = 1007;

   int MENU_BAR = 1008; //Identità della barra dei menu

   int SET_MENU_BAR = 1009; //Messaggio per MAIN_WINDOW

   int SET_LOCALE = 1010; //SET_LOCALE, String (identifica il linguaggio richiesto)

   /** Una mappa di stringhe */
   class StringMap extends java.util.HashMap<String, String> {};
}
```

Da notare che la parola SET_LOCALE è parte della forma richiesta per esprimere il significato “imposta un certo linguaggio come lingua corrente”. La richiesta completa necessita di un simbolo che definisca quale linguaggio debba essere “caricato”. Questo fatto richiede un ulteriore simbolo comunemente noto. Ne vediamo due in MenuBar:

```
package sample;

import it.tukano.ec.*;
import it.tukano.ec.basic.*;
import static sample.Language.*;
import javax.swing.*;

public class MenuBar extends BasicEntity {
   private JMenuBar menuBar = new JMenuBar();
   private JMenu file = menuBar.add(new JMenu());
   private JMenu lang = menuBar.add(new JMenu());
   private JMenuItem open = file.add(new JMenuItem());
   private JMenuItem close = file.add(new JMenuItem());
   private JCheckBoxMenuItem ita = (JCheckBoxMenuItem)lang.add(new JCheckBoxMenuItem());
   private JCheckBoxMenuItem eng = (JCheckBoxMenuItem)lang.add(new JCheckBoxMenuItem());

   public MenuBar() {
      super(MENU_BAR);
      ButtonGroup langGroup = new ButtonGroup();
      langGroup.add(ita);
      langGroup.add(eng);
      eng.setSelected(true);
      ita.addActionListener(new Runfaref(this, "setLanguage", "ita"));
      eng.addActionListener(new Runfaref(this, "setLanguage", "eng"));
   }

   public void processComm(Comm c) {
      if(c.hasMessage(SET_CHANNEL) && getChannel() != null) {
         requestLabels();
      } else if(c.hasMessage(TEXT_MAP)) {
         StringMap map = c.getReferent(StringMap.class);
         if(map != null) {
            new Runfaref(this, "setLabels", map).invokeLater();
         }
      }
   }

   public void catchComm(Comm c) {

   }

   public void setLanguage(String loc) {
      Comm c = new Comm(getId(), RESOURCE_MANAGER, SET_LOCALE, loc);
      sendComm(c);
   }

   public void setLabels(StringMap map) {
      file.setText(map.get("menu.file"));
      lang.setText(map.get("menu.language"));
      open.setText(map.get("menu.open"));
      close.setText(map.get("menu.close"));
      ita.setText(map.get("menu.ita"));
      eng.setText(map.get("menu.eng"));
      sendSetMenuBarComm();
   }

   private void sendSetMenuBarComm() {
```

```
      Comm c = new Comm(getId(), MAIN_WINDOW, SET_MENU_BAR, menuBar);
      sendComm(c);
   }

   private void requestLabels() {
      Comm c = new Comm(getId(), RESOURCE_MANAGER, TEXT_MAP, null);
      sendComm(c);
   }
}
```

In risposta alla pressione di uno dei pulsanti della lingua, MenuBar “dice” SET_LOCALE a RESOURCE_MANAGER. Come sempre possiamo compilare ed eseguire. Non vedremo alcun risultato utile perchè manca la comprensione del significato espresso da MenuBar con quella comunicazione. Stabiliamo, come predetto, che ResourceManager capisca questo messaggio.

```
package sample;

import it.tukano.ec.*;
import it.tukano.ec.basic.*;
import static sample.Language.*;
import java.util.*;

public class ResourceManager extends BasicEntity {
   private volatile String currentLocale = "eng";
   private Map<String, StringMap> localeMap = new HashMap<String, StringMap>();

   public ResourceManager() {
      super(RESOURCE_MANAGER);
      StringMap engMap = new StringMap();
      engMap.put("window.title", "Sample - MainWindow");
      engMap.put("menu.file", "File");
      engMap.put("menu.language", "Language");
      engMap.put("menu.open", "Open");
      engMap.put("menu.close", "Close");
      engMap.put("menu.ita", "Italiano");
      engMap.put("menu.eng", "English");
      localeMap.put("eng", engMap);

      StringMap itaMap = new StringMap();
      itaMap.put("window.title", "Sample - Finestra principale");
      itaMap.put("menu.file", "File");
      itaMap.put("menu.language", "Linguaggio");
      itaMap.put("menu.open", "Apri");
      itaMap.put("menu.close", "Chiudi");
      itaMap.put("menu.ita", "Italiano");
      itaMap.put("menu.eng", "English");
      localeMap.put("ita", itaMap);
   }

   public void processComm(Comm c) {
      if(c.hasMessage(TEXT_MAP)) {
         sendTextMap(c.getFrom());
      } else if(c.hasMessage(SET_LOCALE)) {
         String locale = c.getReferent(String.class);
         if(locale != null) {
            setLocale(locale);
         }
      }
   }

   public void catchComm(Comm c) {

   }

   private void notifyTextMap() {
      Comm comm = new Comm(getId(), ANYONE, TEXT_MAP, localeMap.get(currentLocale));
      sendComm(comm);
   }

   private void sendTextMap(int reqId) {
      Comm comm = new Comm(getId(), reqId, TEXT_MAP, localeMap.get(currentLocale));
      sendComm(comm);
   }

   private void setLocale(String code) {
      if(localeMap.get(code) != null) {
```

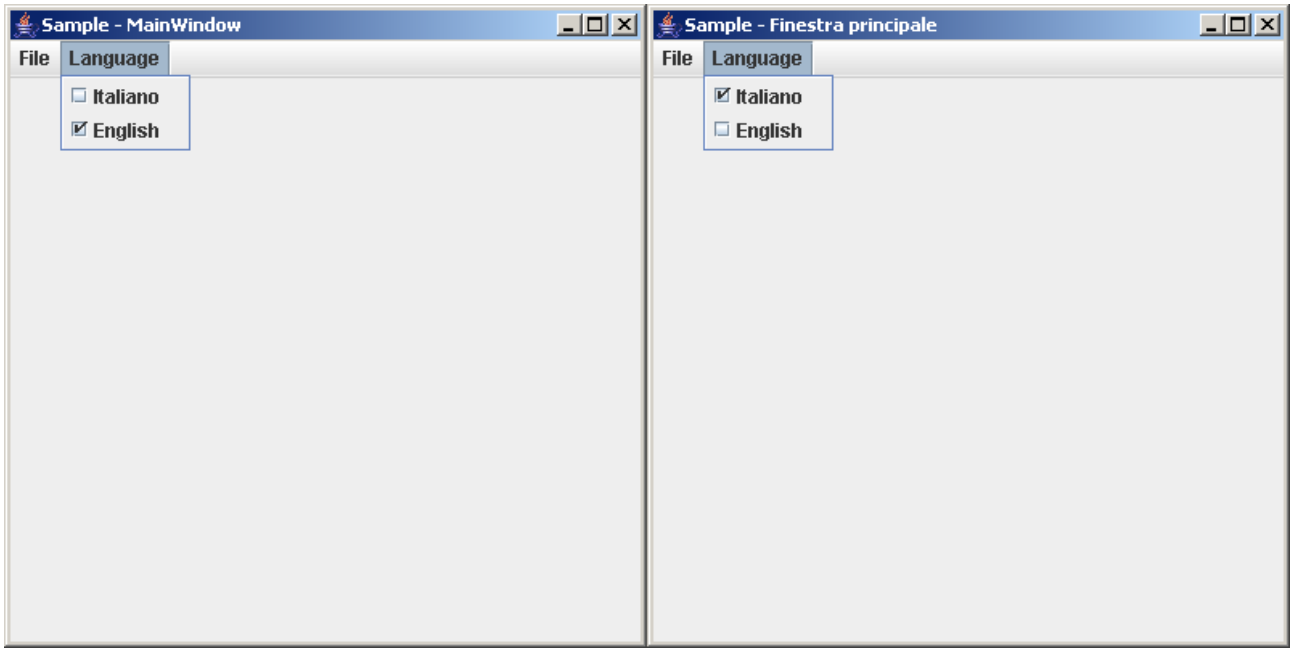
```
            currentLocale = code;
            notifyTextMap();
         }
      }
   }
```

Ora ResourceManager possiede, sbrigativamente, più di un insieme di stringhe, associate a diversi codice di località – due, eng e ita. Quando qualcuno dice a ResourceManager SET_LOCALE, ResourceManager cambia, se possibile, la località di base ed esclama ANYONE TEXT_MAP. Qualcuno in grado di comprendere questo messaggio c'è già.

```
package sample;

import it.tukano.ec.*;
import it.tukano.ec.basic.*;
import static sample.Language.*;
import javax.swing.*;

public class MainWindow extends BasicEntity {
   private JFrame window = new JFrame();

   public MainWindow() {
      super(MAIN_WINDOW);
      window.setSize(400, 400);
      window.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
   }

   public void catchComm(Comm c) {
      if(c.hasMessage(TEXT_MAP) && c.hasFrom(RESOURCE_MANAGER) && c.hasTo(ANYONE)) {
         new Runfaref(this, "setLabels", c.getReferent(StringMap.class)).invokeLater();
      }
   }

   public void processComm(Comm c) {
      if(c.hasMessage(SET_CHANNEL) && c.getReferent(Entity.class) != null) {
         new Runfaref(this, "showWindow").invokeLater();
         requestLabels();
      } else if(c.hasMessage(TEXT_MAP) && c.hasFrom(RESOURCE_MANAGER)) {
         new Runfaref(this, "setLabels", c.getReferent(StringMap.class)).invokeLater();
      } else if(c.hasMessage(SET_MENU_BAR)) {
         JMenuBar menuBar = c.getReferent(JMenuBar.class);
         if(menuBar != null) {
            new Runfaref(this, "setMenuBar", menuBar).invokeLater();
         }
      }
   }

   public void setMenuBar(JMenuBar mb) {
      window.setJMenuBar(mb);
      window.validate();
   }

   public void showWindow() {
      window.setVisible(true);
   }

   public void requestLabels() {
      Comm req = new Comm(getId(), RESOURCE_MANAGER, TEXT_MAP, null);
      sendComm(req);
   }

   public void setLabels(StringMap map) {
      window.setTitle(map.get("window.title"));
   }
}
```

Ricordo che, secondo la distribuzione standard della classe utile BasicEntity, in catchComm finiscono i messaggi dispersi nell'ambiente che non abbiano come destinatario l'entità che definisce il metodo in questione. è come dire che MainWindow sente RESOURCE_MANAGER dire TEXT_MAP, percependo inoltre l'assenza di un destinatario univoco del messaggio (ANYONE). MainWindow sente anche tutti gli altri messaggi che RESOURCE_MANAGER invia. Ad esempio ascolta quanto ResourceManager dice a MenuBar TEXT_MAP. Ma decide di non fare nulla essendo

istruito a non manipolare messaggio quando sia chiaro che il destinatario non è sé stesso. Una questione di educazione. Poiché MainWindow comprende un messaggio TEXT_MAP senza destinatari e ResourceManager invia un TEXT_MAP senza destinatari, nel sistema si produrrà la reazione voluta da MainWindow. Nel nostro caso, il titolo della finestra assume un valore diverso dopo la pressione dei pulsanti della località nella barra dei menu.



Se può farlo MainWindow allora può farlo anche MenuBar.

```
package sample;

import it.tukano.ec.*;
import it.tukano.ec.basic.*;
import static sample.Language.*;
import javax.swing.*;

public class MenuBar extends BasicEntity {
   private JMenuBar menuBar = new JMenuBar();
   private JMenu file = menuBar.add(new JMenu());
   private JMenu lang = menuBar.add(new JMenu());
   private JMenuItem open = file.add(new JMenuItem());
   private JMenuItem close = file.add(new JMenuItem());
   private JCheckBoxMenuItem ita = (JCheckBoxMenuItem)lang.add(new JCheckBoxMenuItem());
   private JCheckBoxMenuItem eng = (JCheckBoxMenuItem)lang.add(new JCheckBoxMenuItem());

   public MenuBar() {
      super(MENU_BAR);
      ButtonGroup langGroup = new ButtonGroup();
      langGroup.add(ita);
      langGroup.add(eng);
      eng.setSelected(true);
      ita.addActionListener(new Runfaref(this, "setLanguage", "ita"));
      eng.addActionListener(new Runfaref(this, "setLanguage", "eng"));
   }

   public void processComm(Comm c) {
      if(c.hasMessage(SET_CHANNEL) && getChannel() != null) {
         requestLabels();
      } else if(c.hasMessage(TEXT_MAP)) {
         StringMap map = c.getReferent(StringMap.class);
         if(map != null) {
            new Runfaref(this, "setLabels", map).invokeLater();
         }
      }
   }

   public void catchComm(Comm c) {
```

```
      if(c.hasMessage(TEXT_MAP) && c.hasFrom(RESOURCE_MANAGER) && c.hasTo(ANYONE)) {
         new Runfaref(this, "setLabels", c.getReferent(StringMap.class)).invokeLater();
      }
   }

   public void setLanguage(String loc) {
      Comm c = new Comm(getId(), RESOURCE_MANAGER, SET_LOCALE, loc);
      sendComm(c);
   }

   public void setLabels(StringMap map) {
      file.setText(map.get("menu.file"));
      lang.setText(map.get("menu.language"));
      open.setText(map.get("menu.open"));
      close.setText(map.get("menu.close"));
      ita.setText(map.get("menu.ita"));
      eng.setText(map.get("menu.eng"));
      sendSetMenuBarComm();
   }

   private void sendSetMenuBarComm() {
      Comm c = new Comm(getId(), MAIN_WINDOW, SET_MENU_BAR, menuBar);
      sendComm(c);
   }

   private void requestLabels() {
      Comm c = new Comm(getId(), RESOURCE_MANAGER, TEXT_MAP, null);
      sendComm(c);
   }
}
```

Concludendo.

Credo che trattare della comunicazione tra oggetti in via esplicita renda più chiaro il significato di alcune locuzioni tipiche del modo in cui si definisce l'orientamento agli oggetti. Il contratto dichiarato in una classe è più di un elenco di metodi e campi pubblicamente accessibili. È una lingua. È, precisamente, la lingua che un oggetto dichiara di essere in grado di capire. Funziona nello stesso modo in cui ho creduto di presentare la questione della comunicazione: è un insieme di simboli che regge l'esistenza del sistema e, qualora sia rispettato, prelude al funzionamento dell'oggetto. Prelude ma non è sufficiente. Affinchè le cose producano un risultato occorre che la lingua espressa nel contratto sia compresa da coloro che vogliano interagire con quell'oggetto. La comprensione non è solo una questione di conoscenza dei simboli ma è anche e, forse, soprattutto, comprensione del significato espresso in tali simboli. Nel dialogo tra entità che ho ritenuto utile presentare, ogni entità conosce tutti i simboli usabili per comunicare. Quei simboli corrispondono, in effetti, all'insieme dei contratti pubblici di un sistema ordinariamente orientato agli oggetti. Pur conoscendo tali simboli, nessuna entità è stata in grado di espletare correttamente il proprio compito finchè l'uso delle parole non è divenuto conforme al significato implicito di quei simboli.

In un sistema orientato agli oggetti non si invia un messaggio ad un oggetto: non sarebbe sufficiente ad alcunchè. In un sistema orientato agli oggetti si inviano dei significati espressi in parte dai messaggi inviato e in parte da un accordo più generale sulle capacità e sui comportamenti dei singoli partecipanti al sistema.

Dovrebbe inoltre essere chiaro il perchè si parli di necessità di una buona definizione dei contratti. Come l'insieme di costanti appartenenti alla lingua parlata dalle entità presentate, la persistenza sia dei simboli del contratto di ciascun oggetto sia del significato di tali simboli è il fondamento della capacità di ogni oggetto di esistere, attraverso la sua identità simbolica, e di agire, non potendo produrre alcun effetto su altre parti del sistema senza sapere quali parole usare per far sentire la propria voce. Non si fa affidamento sulla costanza del nome di un metodo perchè altrimenti gli altri non compilano più. Certo deve esistere una costanza del simbolo. Ma deve esistere soprattutto una costanza del significato espresso in quel simbolo. Farne una questione di varianti e invarianti può

essere geometricamente interessante ma, credo, rende poco l'idea del perchè sia necessario esprimere le condizioni che consentono ad un oggetto di operare secondo una certa capacità. È certamente possibile e, si è visto, anche piuttosto facile, garantire l'autonomia di un oggetto, cioè garantire che un'entità esista a prescindere dalle mutazioni che intervengono nell'ambiente in cui si trova. Ma, concordemente alla grammatica, ritengo impossibile far sì che un oggetto possa interagire con altri oggetti senza che tale interazione sia fondata su una forma convenzionale di espressione delle proprie intenzioni.

Note.

Il framework di comunicazione presentato è totalmente ignaro di questioni relative alla concorrenza. La parte certamente più sensibile è BasicChannel: propaga messaggi ad entità senza considerare la possibilità che queste, per effetto di un receiveComm concorrente, possano venir meno a metà dell'opera di notifica. Il problema è risolvibile in modo banale. Un ConcurrentChannel potrebbe tranquillamente accodare i Comm ricevuti in una coda la cui consumazione sia affidata ad un Thread ad hoc. poiché la rimozione di entità dal canale avviene attraverso un Comm tale rimozione sarebbe sempre in sincronia con la propagazione delle comunicazione. In questo contesto l'unico avvertimento da fornire sarebbe quello di non intasare il procedimento di ricezione ed invio delle comunicazioni con istruzioni la cui esecuzione possa richiedere troppo tempo.