OpenSpaces

An Object-Oriented Framework for Heterogeneous Coordination

ESUG Summer School 2000

Thomas F Hofmann

University of Berne, Switzerland

hofmann@iam.unibe.ch

Heterogeneous Coordination

or: what's this framework for?

„Coordination is managing dependencies between activities“. This means activities that are performed by individual processes or agents which need to synchronize to accomplish their tasks.

Heterogeneity requirements for coordinating agents in *distributed open systems* are:

- Being able to deal with several platforms
- Being usable with multiple programming languages.

OpenSpaces has these properties and offers a set of highly configurable coordination models.

Communication

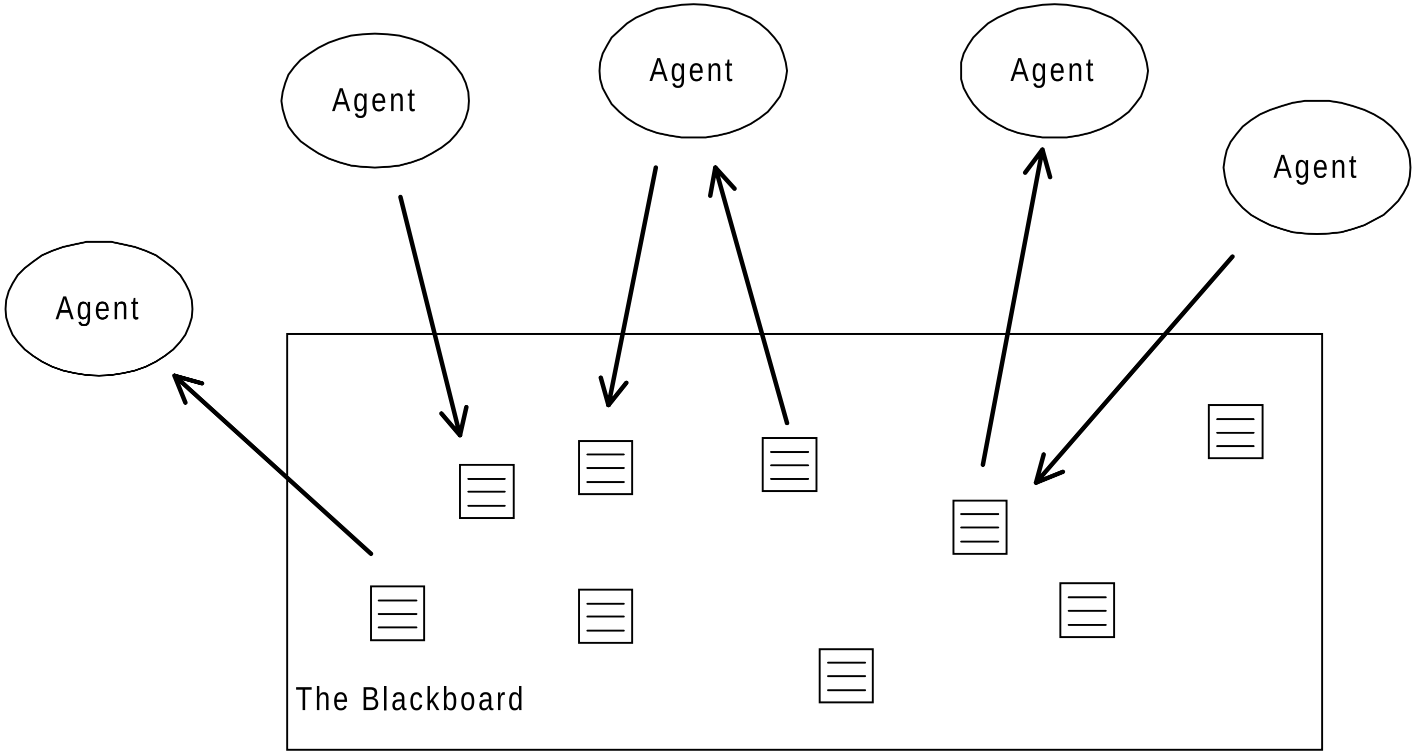
To cooperate, the processes need to exchange informations. Communication paradigms like message passing, RPC or RMI all have the disadvantage of **tight coupling**:

- The sender of a message must know the exact *identity and address* of a receiver.
- Need for *synchronization*: the sender must wait for the receiver to be ready for communicating.

In open systems this tends to be too restrictive. One solution is the concept of **generative communication** which was introduced with the **coordination language Linda.** [Carriero and Gelernter, ‘86].

Blackboard Architecture

The blackboard is a shared repository where agents can exchange „stuff“, which can be messages, calculation results, tasks, etc.



Linda (I)

Tuple A vector of typed elements

Tuple Space The shared repository for tuples

Operation primitives:

- **out (aTuple)** *Write* aTuple to the space.
- **in (aTemplate)** *Take* a tuple from the space that matches with aTemplate.
- **rd (aTemplate)** *Read:* get a copy of a matching tuple, don’t remove it.
- eval (anActiveTuple) This creates a process at the space that results in a passive tuple.

Linda (II)

The retrieval of tuples is *associative*. A **template** is a tuple with 0..n *wildcards* that acts as a *mask* to specify the kind of tuples the caller is interested in.

A tuple *matches* a template if:

- both have the same number of fields
- the types of their corresponding fields are equal
- every actual field of the tuple has the same value as the corresponding field of the template

Wildcards are denoted with a '**?**' followed by the variable to which the corresponding value of a found tuple will be bound.

Using Linda

Examples:

```
out('hello', 'world', 123, 3.14);
int i; float f;
rd('hello', 'world', ?i, ?f);
```

succeeds and binds 123 to **`i`** *and 3.14 to* **`f`**

```
in('hello', 'world', ?i, ?f);
```

succeeds and removes tuple from Space

```
rd('hello', 'world', ?i, ?f);
```

does NOT succeed ...

Properties of Linda

Associative Addressing

Agents specify *what* data they need, not where to find it.

Non-Determinism

It is not known which tuple will be retrieved if there are more than one at the space that would match the template.

Uncoupling of Agents

Agents do not need to know about each other's identity or location, only the Tuple Space must be known. They also do not have to synchronize to communicate, not even simultaneous existence is needed.

Expressiveness

It has been shown that Linda is capable to express a large class of parallel and distributed algorithms.

[Carriero, Gelernter 90]

With a suitably choosen design all typical architectural styles may be realized.

E.g. **master-worker:**

- The master writes to the space a set of tuples to be worked on.
- All workers repeatedly take task-tuples, do their job with it and put the resulting tuple back to the space.

Design aspects

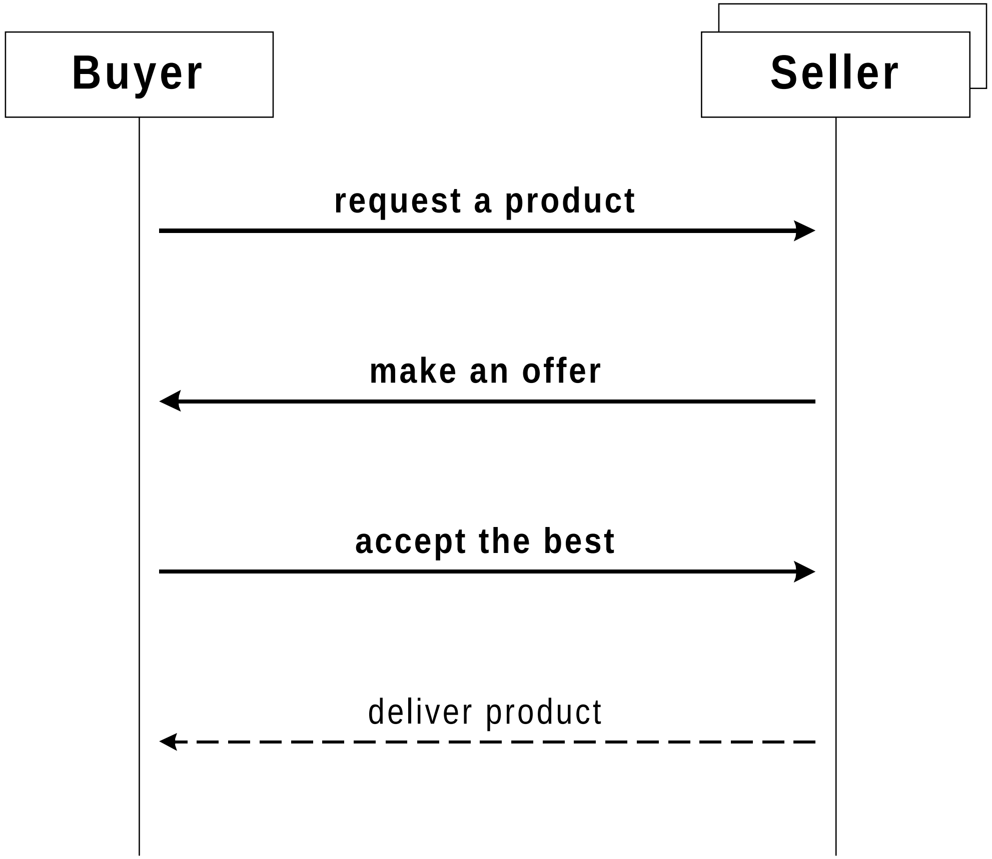
Many implementations have been developed since the original Linda. They introduced extensions like:

- Multiple tuple spaces
- Tuples as objects
- New operations like e.g. bulk-retrieving
- New matching strategies
- Rules gouverning the spaces semantics
- Transactions, distributed events

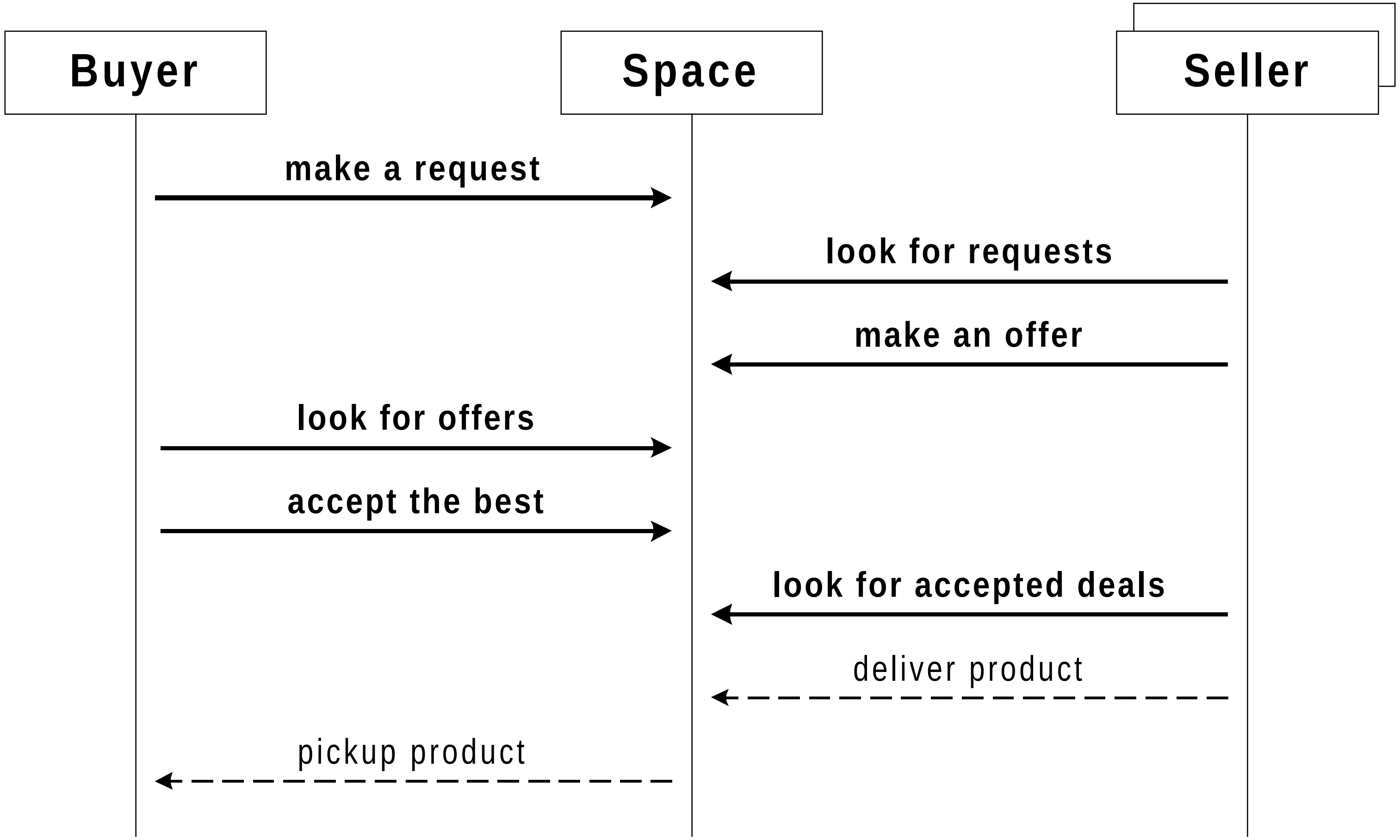
OpenSpaces is designed to be most extensible and configurable to offer every application a mass tailored coordination language.

A Market Place Example

A simple protocol of a typical trading situation:



The Space Adaptation



Analysis

- Two kinds of actors: **Buyer** and **Seller**.
- Space **entries** to represent **requests**, **offers** and **deals.**
- Requests describe a wanted product or service.
- Offers *reference* a request and specify the price.
- Deals reference a deal.
- Requests must be readable by anyone interested.
- Offers must be readable (only) for the buyer who issued the referenced request.
- Deals must be readable (only) for the seller.

Stepping through a Trade

As actual participants we specialize **`SpaceAgent`** to **`Buyer`** and **`Seller`**. Their respective protocols support the role specific actions of their respective parts:

`Buyer>>makeRequest` Append a new request form.

`Seller>>collectRequests` Scan for newly arrived requests.

`Seller>>makeOffer` Append a new offer form referencing a request by its index.

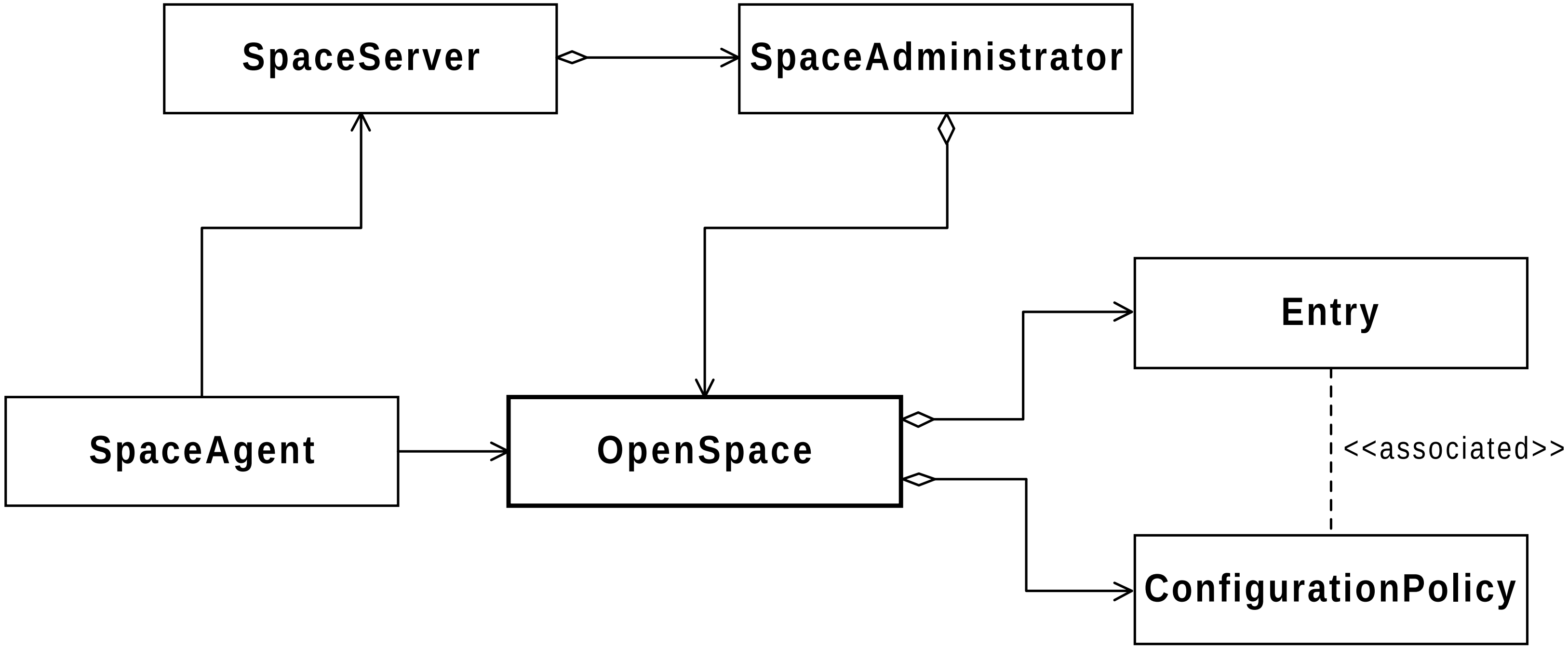
`Buyer>>collectOffers` Scan for offers to ownrequest(s) and take them from the market.

`Buyer>>acceptOffer` Append a deal form referencing the offer by its index. Remove the referenced request.

`Seller>>collectDeals` Scan for deals accepting own offers and remove them.

Open Spaces

The core classes of the framework:



OS 1: Class Entry

Entries contain the data to be exchanged. To define suitable attributes each application has to subclass the (empty) root class **`Entry`**. Any single objects or collections may be used as instance variables.

The class of a concrete **`Entry`** descendant also forms the key to associate the entry with a policy object. This **`Configuration Policy`** defines the Space's semantics affecting instances of the associated entry classes. This includes the used *matching algorithm.*

OS 2: Class OpenSpace

`OpenSpace` is the abstraction of the blackboard medium. It holds a collection of entries and offers several ways of accessing it.

The standard operation primitives:

`write: anEntry`	write anEntry to the space
`read: aTemplate`	^ a matching entry or nil
`take: aTemplate`	^ a matching entry or nil

The blocking an bulk-retrieving variants:

`blockingRead: aTemplate`	blocks until success
`readAll: aTemplate`	^ *all* matching entries

Analogous for **`take`**

OS 3: Class SpaceAgent

`SpaceAgent` is the standard abstraction for clients of the space. Space agents hold a reference to their current host space which they get from the globally accessible space server.

The class **`SpaceAgent`** is often subclassed to add application specific behavior and to hide the underlying communication structures.

OS 4: Getting a Space

The **`SpaceServer`** is a *name server* used by all space agents to access a space. Spaces are looked up by their name, they must be registered to become available.

If a request is made specifying an unknown space name, the space server may act as a *factory*. It can create and register a new space with the given name.

The space server delegates the actual managing of the space references to the **`SpaceAdministrator`** and redirects the allowed requests to it.

Forms

In the Market Place example requests, offers and deals are represented with the **`Entry`** subclass **`Form`**. Its sole attribute is a dictionary, called **`bindings`**, to hold any key-value binding. This offers flexibility since additionally needed values can be added without subclassing.

A form matches a template if:

- The form is an instance of the same class as the template or of a subclass
- The forms bindings contain all the keys of the templates bindings
- Their respective values are equal

Additional keys are not considered.

Form Matching

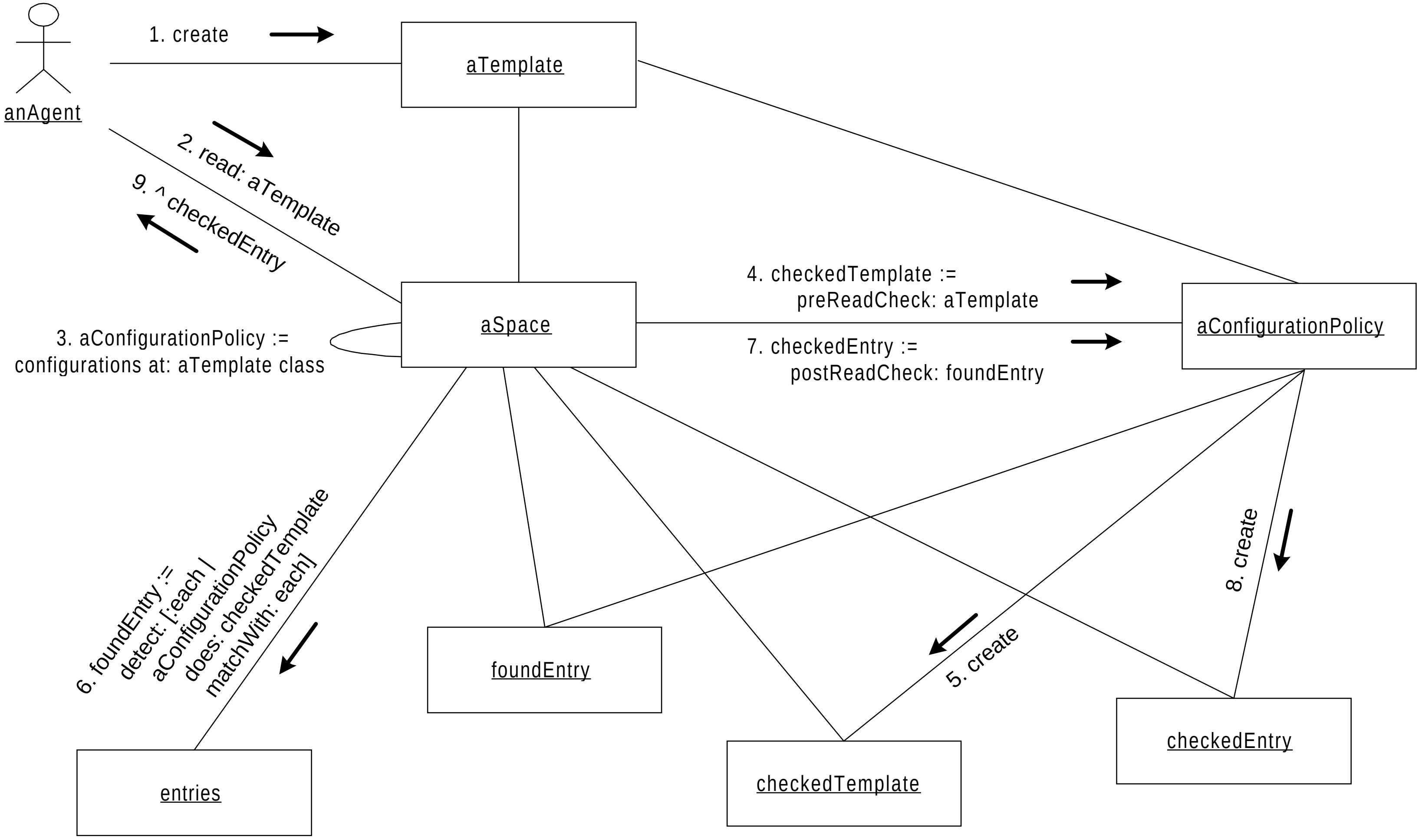
```
FormPolicy >> does: aForm matchWith: aTemplate
  "Answer true if aForm contains all keys of aTemplate
   and all respective values are equal."
  |ok|
  ok := (aForm isKindOf: aTemplate class)
          and: [aTemplate bindings notNil].
  ok ifTrue:
      [aTemplate bindings keys
        do: [:key |
          (aForm bindings includesKey: key)
            ifFalse: [ok := false]
            ifTrue:
              [ok := (aForm bindings at: key)
                     = (aTemplate bindings at: key)]]].
  ^ ok
```

OS 5: Configuration Policies

Each entry needs an associated **`ConfigurationPolicy`** which defines the *matching algorithm* to be used for the associative retrieval of such entries from the space.

Additionally the policy defines *hook methods* which are called before and after each of the space operations. Like that every access can be controlled. The hook methods can change or refuse a used entry or template. They can also access the space. Basically any action may be triggered. (!)

The Read Contract



Unique References

Each offer must reference the request it reacts to, each deal the respective offer. Therefore each form gets a binding **#index->anIndex** where **anIndex** should be unique. A specialized **TailEntry** remembers the highest index used so far.

To **append** a form the **MarketAgent** must:

- **take** the tail entry
- increase its index
- **write** it back
- set the forms **#index**-value to the new index
- write the form

Market Place V 2

Δ Consistency of the Market Place does require:

- unique indices
- equal indices of a to be written form and the tail
- correct referencing
- complete forms

⇒ The **`ConfigurationPolicy>>preWriteCheck`** method can assert this: only correct forms pass, others are rejected. Three different subclasses define the method for the respective forms.

`Form` is subclassed to **`Request`**, **`Offer`** and **`Deal`**, which can be associated with the corresponding policies (without any changes in the implementation).

Consistency Checks

`RequestPolicy>>preWriteCheck: aForm`

- check if the forms bindings include the keys **`#section`**, **`#product`** and **`#index`**.
- read the tail entry and check if the forms **`#index`**-value is equal to the tails index and if there is no other form present with this index.
- if OK answer the form, else nil to reject it.

The pre-write checks in **`OfferPolicy`** and **`DealPolicy`** are similar. Additionally they check the references of the forms. The **`DealPolicy`** removes the referenced request form if it still is present.

Market Place V 3

Δ The *index incrementing procedure* is a bit awkward. Doing this at the space would reduce the responsibility of the market agents and also the network traffic.

⇒ In its **`preWriteCheck`** method a specialized **`AutomaticIndexPolicy`** takes care of the tail business and sets the index of the form that should be written.

Since the write operation returns the actually written entry, the space agent can check it for the received index.

Market Place V 4

Δ After some running time it is quite probable that forgotten entries start to clutter a space. Some garbage collection is needed. One solution is adding a timestamp to every entry beeing written to the space. After expiry of its lifetime it will be discarded.

⇒ At the space this can be done with a suitable configuration policy. On **`preWriteCheck`** a binding **`#arrivalTime->(Time now)`** is added to the form. On every call of a retrieving operation the **`preWriteCheck`** discards every outdated form at the space.

Garbage Collection

To discard the outdated forms the pre-access hook perform a **`readAll`**. The received collection is scanned and each expired form is removed from the space by perfoming a **`take`**.

Since the two operations again will call the pre-access-hooks this could cause loops! To distinguish between a client-initiated and a policy-initiated call of the pre-access hook a flag is set before calling the actual cleanup method and unset after it. The pre-access-hooks check this flag to bypass the usual checks if set.

Reconfiguration

The space offers a method to register an entry class with a specific configuration policy and one to cancel such an association. Since the policy is looked up for every access, it is easy to *reconfigure the system on the fly*.

The only thing to consider thereby is how to proceed with the already present entries of the affected class. The configuration policy has a special hook method: **`updateOldEntriesOfClass: aClass`**, which is called right after a new registering of the class with the new policy. Any necessary action for a clean transition can be done there.

CORBA: I3

DST offers the Implicite Invocation Interface (I3) which provides remote communication without explicit IDL definitions. It works only between Smalltalk images with DST loaded, but all CORBA features may be used.

However, to be interoperable with other CORBA compilant languages, we need to declare IDL modules to instruct the ORB how to marshall and unmarshall requests and parameters or results.

It is straight forward to extend a system built using I3 to support any language by adding the IDL.

CORBA: IDL

- Each object which will be called remotely must have an interface definition.
- Each parameter and return value must be an instance of a declared type.
- Every IDL type is recursively built from simple data types like float, short, string, etc.
- Interfaces are transmitted as **remote object references.**
- An **IDL-struct** is passed by value.
- Both participating ORBs must have synchronized **repositories** containing the same IDL modules.

IDL Examples (I)

```
interface OpenSpaceInterface {
  #pragma selector write write:
  any  write(in any anEntry);
  #pragma selector read read:
  any  read(in any aTemplate);
  #pragma selector take take:
  any  take(in any aTemplate);

  (...)

  #pragma selector takeAll takeAll:
  OrderedCollection takeAll(in any aTemplate);
  };
};
```

IDL Examples (II)

```
#pragma class OrderedCollection OrderedCollection
typedef sequence<any> OrderedCollection;

#pragma class Association Association
struct Association{ any key; any value; };

#pragma class Dictionary Dictionary
typedef sequence<Association> Dictionary;

#pragma class Form Form
struct Form {
  Dictionary bindings;
};
```

Conclusion

OpenSpaces is configurable and extensible

- The matching algorithm is arbitrarily definable.
- The access hooks may perform any side-effect.
- White-box style extensions by subclassing and black-box style extensions by setting up configuration policies are possible.

OpenSpaces works heterogeneous

Several server platforms: Visual Works runs on UNIX, windows, mac, Linux.

Clients may be written in any language with an ORB.

Outlook / Extensions

- Distributed transactions
- Authorization for reconfiguring the system
- Remote registering new entry types through the client (IDL description). Needs generic configuration policies.