OpenSpaces: An Object-Oriented Framework For Reconfigurable Coordination Spaces ⋆

Stéphane Ducasse, Thomas Hofmann, and Oscar Nierstrasz

Software Composition Group, Institut für Informatik (IAM), Universität Bern
(ducasse,hofmann,oscar)@iam.unibe.ch
http://www.iam.unibe.ch/∼(ducasse,hofmann,oscar)/
Published In *Coordination Languages and Models, LNCS 1906, pp. 1-19, 2000*.

Abstract. Tuple spaces have turned out to be one of the most fundamental abstractions for coordinating communicating agents. At the same time, researchers continue to propose new variants of tuple spaces, since no one approach seems to be universally applicable to all problem domains. Some models offer a certain configurability, but existing approaches generally stop at a fixed set of configuration options and static configuration at instantiation time. We argue that a more open approach is needed, and present OPENSPACES, an object-oriented framework that supports static configurability through subclassing across several dimensions, as well as dynamic configurability of policies through run-time composition. We introduce OPENSPACES by showing how it can be used to instantiate a typical application, and we present an overview of the framework, implemented in Smalltalk, detailing the various degrees of configurability.

Keywords: Object-Oriented Languages, Frameworks, White Box Reuse, Black Box Reuse, Dynamic Reconfiguration

1 Introduction

Tuple spaces have proven to be among the most fundamental and successful abstractions for coordinating concurrent activities. There are numerous reasons why this should be so, both technical and pragmatic.

On the technical side, tuple spaces have the advantage of capturing both communication and synchronisation in a simple and natural way. Tuples themselves represent resources that can be communicated, shared and exchanged, without the need to use additional synchronisation mechanisms. Furthermore, associative lookup obviates the need for communicating agents to be explicitly aware of one another's "identity."

On the pragmatic side, there are many kinds of problems that map naturally to the tuple space view of the world, namely, that there are many different kinds of concurrent "agents" that want to exchange "stuff" with one another. Tuples, in a sense, represent the least common denominator of data structures, and can therefore be used to easily model almost any kind of "stuff."

Why, then, have so many different variants of tuple spaces appeared over the years? Numerous variations on tuple spaces have been proposed, and it does not seem as

⋆ This research is funded by the Swiss Government under Project no. NFS-2000-46947.96.

though this proliferation will end soon [PA98]. Again, there are numerous reasons why this should be so, both technical and pragmatic.

On the pragmatic side, applications requirements impose different policies governing what kinds of "stuff" are exchanged, which agents have access to which resources, how resources are matched against queries, and what kinds of actions may be triggered upon exchanges of resources. On the technical side, tuple spaces say nothing about how such policies may be introduced as higher-level abstractions. (Linda, for example, provides no abstraction mechanisms whatsoever, considering that to be a matter for the host language.)

To alleviate this problem, some researchers have proposed tuple spaces with various configurable parameters [Tol97]. Still, the degree of configurability in these approaches tends to be limited, and any configuration parameters must be fixed when the tuple space is instantiated. Even approaches that allow different matching algorithms to be employed do not allow these to be changed at run-time. For application domains in which the policies under which agents exchange information and resources may be dynamically negotiated, this is not enough.

We propose OPENSPACES, an object-oriented framework which offers the core services of tuple spaces as standard features, and at the same time allows the policies in place to be arbitrarily tailored, and set in place at run-time. OPENSPACES is both a white-box and black-box framework. Individual tuple space abstractions can be specialised by subclassing, and their instances can be dynamically configured and composed at run-time.

OPENSPACES can be tailored in the following ways: (1) Different kinds of entities to be stored in an `OpenSpace` are defined by subclassing `Entry`. (2) Different matching policies are defined by subclassing `ConfigurationPolicy`. (3) Methods to be triggered before and after every access to a space can be specialized. This is useful for validating, modifying or rejecting entries, or for triggering any useful side effect. (4) These hook methods and the matching algorithms can be plugged in dynamically in a black-box fashion and take effect without restarting the system. (5) A special update method can be triggered to automatically adapt affected entries whenever the policy in place is dynamically changed.

In section 2 we introduce a motivating example that illustrates several of the kinds of configurability typical of coordination applications. In section 3 we introduce OPENSPACES by showing how it can be used to tackle the motivating example. Sections 4 and 5 present the OPENSPACES framework in more detail and illustrate how the framework supports variation. In the last two sections we present related work and conclusions.

2 Example: An Electronic Market Place

We will motivate OPENSPACES with a running example that exhibits not only a classical set of coordination requirements, but also common forms of variability. In the following section we will see how OPENSPACES can fulfill both sets of needs.

2.1 Scenario

Let us consider a typical trading situation: a buyer agent (i.e., someone representing a client) is looking for some goods or services. He wants to inform potential sellers of his needs and publishes his request e.g. as an advertisement in a newspaper. The sellers see the request and may react by offering a concrete bid for it. The buyer agent can choose amongst all received offers and may accept the one that meets his needs best.

This simple scenario exhibits several classical coordination requirements: the buyer doesn't know the sellers in advance; multiple potential sellers should be informed of the request; neither simultaneous nor synchronous communication between the buyer and sellers is needed; instead, the request should be *published* in some suitable medium; the request can be withdrawn once it is fulfilled (or expires); multiple buyers may wish to publish requests in thematically related media. This negotiation protocol therefore perfectly matches the characteristics of a blackboard-style architecture. Every step can be performed by posting a corresponding entry to the blackboard. It can be read or taken (consumed) by the receiver. Participants don't have to know each others' location or name, they just need to know where to find the blackboard.

2.2 Analysis

Even this simple scenario poses special requirements for the *policies* in place for the blackboard. A minimal implementation would require the following setup: (1) There are two kinds of agents to represent the buyers and the sellers. (2) The agents exchange entries (possibly, but not necessarily tuples) representing requests, offers and deals. *Requests* describe the desired product and maximum price. *Offers* describe the offered product and price. A *Deal* finally is written to accept a received Offer. (3) An offer must reference the request it responds to. A deal must reference the offer it accepts. (4) Requests must be readable by anyone interested, but they may only be withdrawn by the issuing buyer, when he is no longer interested in receiving more offers. (5) Offers referencing a request may only be read and removed by the initiating *Buyer*. (6) Deals may be read and removed by the seller who has issued the referenced offer. (7) To divide the market into multiple thematic sections, all entries must have a label with the name of the section they belong to.

3 The Market Place using OPENSPACES

We will now introduce OPENSPACES by showing how it can be used to implement the market place scenario. We start with the simplest solution that works and incrementally show how we can specialize and extend the framework to provide new features or more refined solutions. First we present an overview of the framework to introduce the different entities involved.

3.1 OPENSPACES in a Nutshell

Our framework is tailored to instantiate data-driven coordination languages. The core defines a blackboard style medium, an `OpenSpace`, which allows agents to interact

by exchanging data that are encapsulated as `Entry` objects. The agents are specializations of `SpaceAgent`. To get a reference to an `OpenSpace`, the agent calls the globally accessible `SpaceServer`, a name server that holds a `SpaceAdministrator` maintaining a collection of currently registered spaces.

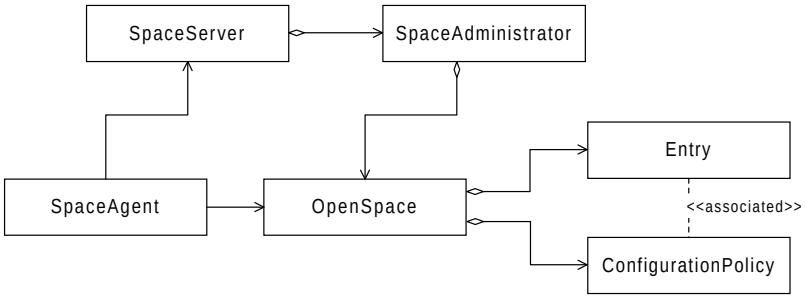

Fig. 1. Overview on the core classes of OPENSPACES.

The space offers the standard accessing primitives: `write: anEntry` puts the entry into the space, `read: aTemplateEntry` and `take: aTemplateEntry` retrieves an entry by associative lookup. The *template* is an entry that is used as a mask to select an entry from the space. The *matching algorithm* determines whether or not an actual entry matches the template and may be returned for it.

In OPENSPACES the matching algorithm is variable. It can be specified as needed for an application. Every used subclass of `Entry` may have its own strategy, which is defined in an associated `ConfigurationPolicy`. In addition the policy object *controls the access to the space* for entries of its associated class. This is realized with validating methods that are applied before and after all accesses. These methods may basically trigger any action on attempts to access the space (cf. section 4.3).

3.2 Market Place V.1: Standard Implementation

Instead of communicating tuples, we will use `Form` as a concrete subclass of `Entry`. It has as its sole attribute a dictionary, called `bindings`, which is used to store associations of any keys and values. This is a flexible approach since additionally needed values may be added without the need to define new subclasses.

A template matches a `Form`: (1) if the form is an instance of the same class as the template or of a subclass, and if (2) the form's bindings contains all the keys of the template's bindings and (3) their respective values are equal. Additional keys of the form are not considered.

The definition of the matching strategy is a responsibility of configuration policy objects. Figure 2 shows the implementation of the matching algorithm for the used forms. It is defined in class `FormPolicy`, extending `ConfigurationPolicy`.

Using the Market Place forms In the forms' bindings we enter the needed data for the trade communication and the name of the section of the market the form belongs to. As mentioned, a `Form` may be a request, an offer or a deal. Therefore it gets a value denoting its type. A request form can have bindings for a product name, for a

```
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
```

Fig. 2. The matching algorithm for forms defined in class `FormPolicy`.

description of the product, and for the maximum price the buyer is willing to pay for it. An offer form may describe a concrete product and price, and holds a reference to the request. A deal form has a reference to the offer it accepts.

3.3 Uniquely Identifiable Entries

To have a uniquely referenceable index for each form, we add a binding for the index and its value to every form when writing the form to the space. To remember the highest index used so far, there is a `Tail` entry to hold its current value. A `MarketAgent` – a specialized `SpaceAgent` – who wishes to append a new form to the market place, takes the tail entry, increments its index value, puts it back to the space and writes the form with the new index to the space (See figure 3). Beside remembering the last used index, the `Tail` entry – actually a counter – acts also as a semaphore, ensuring that only one form is written per index.

```
MarketAgent >> append: aForm
 "Get a new index, add it to aForm and write aForm to the space"
 |template tail|
 template := TailEntry new.
 template section: (aForm bindings at: #section)).
 tail := self hostSpace take: template.
 tail index: tail index + 1.
 self hostSpace write: tail.
 aForm bindings at: #index put: tail index.
 self hostSpace write: aForm
```

Fig. 3. The MarketAgent's appending method.

A `MarketAgent` who wants to read all present requests would have to iterate over all indices up to the last one denoted by the tail. As a simplification the agent may

use the `readAll` operation provided by OPENSPACES, which is similar to `read`, but returns a collection with ALL entries from the space that match the template.

3.4 Stepping through a trade

As actual participants we specialize `MarketAgent` to `Buyer` and `Seller`. Their respective protocols support the role specific actions of their respective parts. These are as follows: (1) The buyer makes a request by appending a new request form to the market space. (2) Sellers get the request when scanning for newly arrived requests. (3) Each seller may make an offer for it by appending an offer form that references the request with its index. (4) The buyer scans for offers to her request and takes them from the market. (5) When detecting a valuable offer, the buyer accepts it by appending a deal form with a reference to the offer and also the initial request to avoid differences. She then removes referenced requests. (6) The seller detects the deal form by scanning for any reactions to its own offers and removes it.

For each participant we have built a simple user interface. The `BuyerUI` allows a user to specify a requested product and send a request form to the space. The UI shows the received index for the request. The UI shows the collected offers and lets the user select the best and send a deal form. The seller has a similar UI for her counterpart.

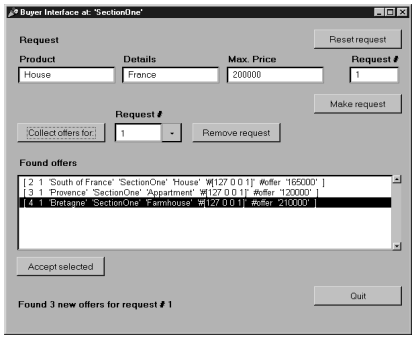

Fig. 4. The user interface for the buyer.

3.5 Market Place V.2: Consistency Assertions

In the first version of the market place, we defined a special matching algorithm for our market forms. In addition to the specific form matching algorithm we now define *access control assertions* in the configuration policy, which are applied at each attempt to read, take or write certain entries. We thereby check the forms for completeness and correctness.

This checking is made in subclasses of `ConfigurationPolicy`. To become effective the policies must be registered with the class of entries it will be affecting. We therefore subclassed `Form` to `Request`, `Offer` and `Deal` (without any changes in the

implementation). The corresponding policies each override the `preWriteCheck:` method to specify the needed checks on the used forms before they are written to the space. The forms are accepted only if they are correct, otherwise they are rejected. (In section 4.3 is a more detailed discussion of these hook methods).

```
RequestPolicy >> preWriteCheck: aForm
 "Check aForm for necessary keys, if its index is equal to the
 tail's and if there is no other form present with this index."
 |ok|
 "... ok := aForm includes keys #section, #product, and #index"
 ok ifTrue:
      [ |template tail|
      template := TailEntry new.
      template section: (aForm bindings at: #section).
      tail := self hostSpace read: template.
      ok := (aForm bindings at: #index) = tail index ].
 ok ifTrue:
      [ template := Form new.
      template bindings
        at: #section put: (aForm bindings at: #section).
      (hostSpace readAll: template)
        do: [:each |
          (each bindings at: #index)=(aForm bindings at: #index)
            ifTrue: [ok := false]]].
 ^ok
   ifTrue: [aForm]     "passed"
   ifFalse: [nil]      "reject"
```

Fig. 5. The `preWriteCheck:` method of the `RequestPolicy` for consistency assertion.

The `RequestPolicy` checks if the request form is complete and if its index actually is unique and equal to the tail entry's index. Figure 5 shows this consistency check. The `OfferPolicy` checks if the offer includes a reference to a request that is actually (still) present at the space. The `DealPolicy` checks if the deal includes references to an offer and a request. If the referenced forms are still at the space they are removed (space clean up).

With the same approach we may increase consistency of the system more by e.g. not allowing an agent to write the same form twice, etc. Note that the form and policy classes can be used with the new policies as soon as they are defined and registered at the space.

3.6 Market Place V.3: Automatic Index Handling

The index incrementing procedure for writing market forms is somewhat awkward. We want to reduce the workload and the responsibility of the market agents - and also the network traffic(!) - by doing this at the space.

We specialize the configuration policies to handle the indices automatically. The agents append forms without checking for a correct index. The policy takes care of the tail business and sets the index of the form accordingly. Since the write operation of the space returns the actually written entry, the agent checks this for the received index.

In Figure 6 the `preWriteCheck:` method passes the validated form back to the space's writing operation where it will be used instead of the original form. Since the

```
AutomaticIndexPolicy >> preWriteCheck: aForm
  "Take the tail entry, increment its index and put it back.
  Enter the new tail index to the form"
  |checkedForm|
  checkedForm := (aForm bindings includesKey: #section)
                  ifTrue: [aForm]
                  ifFalse: [nil].
  checkedForm notNil
    ifTrue:
      [ |template tail newIndex|
      template := TailEntry new.
      template section: (aForm bindings at: #section).
      tail := self hostSpace take: template.
      newIndex := tail index + 1.
      tail index: newIndex.
      self hostSpace write: tail.
      aForm bindings at: #index put: newIndex.
      self hostSpace write: aForm
      checkedForm bindings at: #index put: newIndex].
  ^checkedForm    "return form with new index"
```

Fig. 6. The automatic index handling at the configuration policy

configuration policy is local to the space, this is quickly done and causes less network traffic. It allows us to reduce the responsibilities the agent has to fulfill, allowing it to be 'thinner'. Altogether this modification reduces the risk of errors, improves consistency of the market, and thereby assures even better performance.

3.7 Reconfiguration

As soon as a new configuration policy class is defined and is associated with an entry class, it will be used. I.e. the next space operation with an affected entry will be ruled by the policy's matching algorithm and access checking methods. The important implication here is that the space can be *dynamically* reconfigured. In section 5.4 we present how the framework lets the programmer define the needed procedure to adapt entries that are in the space at the moment the reconfiguration becomes effective.

After these examples of using our framework, we continue with a more detailed description of the classes and contracts in OPENSPACES.

4 The OPENSPACES Framework

OPENSPACES is an object-oriented framework implemented in Smalltalk. It offers the possibility to implement a variety of different data-driven coordination languages to be used in distributed environments. Instantiations may have the characteristics of the original LINDA [GC85] model, of object-oriented approaches like proposed by JAVASPACES [FHA99] and TSPACES [WMLF99]. The novelty of OPENSPACES is that it offers fine-grained configuration options which may also be dynamically changed.

We now present the core classes and their structural relationships, collaborations and the key contracts between these classes.

4.1 Core Classes and their Relationships

The core of OPENSPACES consists of the following six classes:

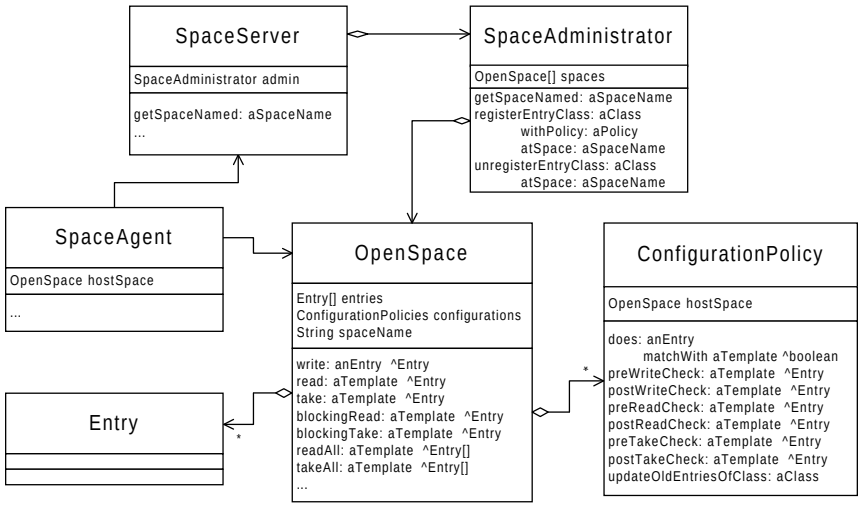

Fig. 7. Structural Relationships of the core of OPENSPACES.

Entries contain the data which space agents may exchange via a space. `Entry` is the abstract root class for all space entries. It has no attributes, applications must define subclasses with the necessary instance variables to hold the exchanged data. There is no restriction concerning number or kind of objects to be held.

The class of a concrete `Entry` descendant forms also the key which is used to associate the class with a `ConfigurationPolicy`. This association governs the behaviour of the Space concerning all operations with instances of the specified entry class. Actually every entry subclass has to be registered at the space with a corresponding configuration policy. Unregistered entry types are rejected.

Configuration Policies represent the semantics of the space's access operations affecting certain classes of entries. Subclasses of `ConfigurationPolicy` define the matching algorithm to be used for retrieving operations and a set of access controlling methods which are applied at each access with the involved entry types.

Open Space is the abstraction representing the blackboard medium. It holds a collection of entries and offers several ways of accessing it. The standard primitives adapted from LINDA are supported: `write`, `read` and `take`. [1] The two retrieving operations `read: aTemplateEntry` and `take: aTemplateEntry` use their parameters as a mask to do an associative lookup of a matching entry. The template is an `Entry` which may have some of its data fields defined and some not. In a general matching strategy the undefined fields act as wildcards for the lookup. Those with actual values restrict the selection of entries that have equal values. The lookup in general is nondeterministic.

The simple `read` and `take` operations are non-blocking, i.e. the calls return immediately, either with a found entry or a null-value, if nothing was detected. OPENSPACES also supports the blocking variants: `blockingRead` and `blockingTake`. These cause the calling client process to suspend until a matching entry is available.

Two additional bulk-retrieving operations are supported. `readAll` and `takeAll` act the same as the simple ones with the exception that they return collections with *all* currently available matching entries. All space access operations are atomic.

The exact behaviour of an `OpenSpace` may be different for any class of used entries, depending on the *configuration policy* objects they are associated with. Therefore `OpenSpace` provides the functionality to manage the *mapping* between entry classes and configuration policies.

Space Agent is the standard user abstraction for the space. It holds a reference to its current space which it gets from the globally accessible `SpaceServer`. The class `SpaceAgent` is often subclassed to add application specific behavior and hide the underlying communication structures.

The Space Server is used by all space agents to access a space. `SpaceServer` is a singleton object that acts as a name server. 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.

The Space Administrator is the actual manager of `OpenSpaces`. The `SpaceAdministrator` holds a collection with references to all currently registered Spaces and implements their accessing by name. It offers the methods to register (and unregister) any local or remote Spaces.

[1] The naming convention was borrowed from JAVASPACES, since it seems more natural and clear to "write" an entry instead of "out-ing" it.

4.2 CORBA as an Implementation Layer

OPENSPACES uses Cincom DST (Distributed Smalltalk) for the distribution. DST is CORBA 2.0 compliant and offers in addition a special feature called 'Implicit Invocation Interface' (I3) which is an extension to the CORBA facilities that provides remote communication between Smalltalk applications without explicit IDL definitions.

Connecting to a Space To get an initial reference to the `SpaceServer` OPENSPACES offers two options for a client. The first is to set its ORB as client to the corresponding address and port number of the server ORB. Like that the naming service of the server ORB can be used to resolve a reference to the space server. The second option is to write an IOR file with the stringified object reference of the space server to an accessible location where the agents may read it and connect themselfes.

Distributed Event Service DST provides an implementation of the CORBA event service protocol. In OPENSPACES this is used for a notification option: a SpaceAgent may subscribe to be notified by the space when an entry is written that matches a given template. The subscription is terminated when a match is found. For continuing notification it can repeatedly be renewed.

Market Place V.4: Automatic Notification As example of notification mechanism, we implemented an automated variant of the Market Place. Buyers would like to be informed of newly arriving Offers the reference their own Requests. Sellers would like to be notified of new Requests and also of Deals for their own Offers.

This variant is easily realized by extending the two agent classes to `NotifiedBuyer` and `NotifiedSeller`. Upon every request the buyer makes, he subscribes at the space with a template of an offer holding the corresponding request number. When notified, he automatically takes the newly arrived offer from ths space. The seller subscribes for requests at initialisation time and for the accepting deals at each offer she issues.

4.3 Framework Contracts

After having briefly described the core classes, we now present how they interact. An important part of the framework's flexibility originates from the contracts of the different access methods available to clients of an `OpenSpace`. We describe in detail the read contract. The take contract is analogous. The write contract differs in some parts.

Read Contract An attempt to perform a `read` operation by an agent initiates the following interactions between the participating entities (cf. figure 8):

1. An agent creates a *template* representing the kind of information he wants to retrieve from the space.
2. The agent calls the space's `read:` method with the template.

3. The space looks up the configuration policy associated with the template class.
4. The space calls the cpolicy's `preReadCheck:` method with the template.
5. The policy creates a *checked template* and returns it.
6. The space iterates over its entries asking the policy for each if it matches with the checked template.
7. The space passes the first *found entry* to the policy's `postReadCheck:` method.
8. The policy creates a *checked entry* and returns it.
9. The space returns the checked entry to the agent.

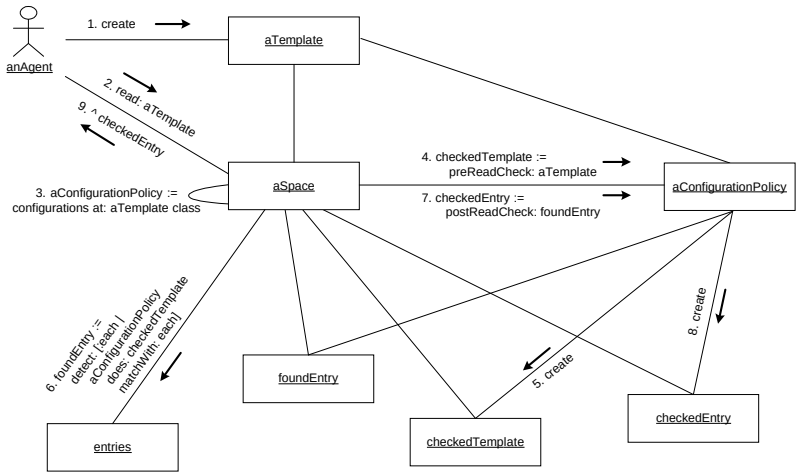

Fig. 8. The read contract: instead of directly accessing the space's entries, the presented template and the found entry are controlled by the configuration policy.

All read variants (`read`, `blockingRead`, `readAll`) apply the same pre- and post-hooks before and after the retrieving, the same holds for the variants of `take`.

Take and Write Contracts The take contract is analogous to the one for read accesses. The difference is that the called hook methods are `preTakeCheck:` and `postTakeCheck:` and a successful lookup results in the removal of the matching entry from the space. The write contract again differs in using its own hook methods. Additionally the write operation has to check the lists with reservations for reading or taking which are maintained for the blocking accesses. This means that it has to scan these two lists to check if the written entry matches with a template used for the blocking accesses. If so the waiting process has to be resumed. The same with all notification handlers. If the new entry matches one of the event handler's templates it has to issue the corresponding event. These checks are performed after the postWriteChecks.

Discussion With the described contract, the hook methods allows a user to specify many useful variants of a space's behaviour. This offer advantages like:

- *Protection* The space can ensure that the entry put in the space holds certain properties and the space agent is ensured that the retrieved entry is coherent.
- *Shifting Responsibility* Instead of requiring the space agents to be responsible for consistency of the space, it is the space itself that does so by invoking the configuration policy that controls its state.
- Reduction of the *network traffic*.
- *Hiding space administration* The space can perform administrative tasks that are hidden from the space agent.
- Basically *any action* may be triggered if needed.

5 Extensions, Configurations and Reconfigurations

The OPENSPACES framework provides three main axes of variability. First, simple subclassing of the core classes allows *white-box extension* and reuse [JF88]. Second, the specialisation of *configuration policy* objects offers *black-box extension*. Moreover, besides compile-time extension, the configuration policy objects allows the runtime *reconfiguration* of a given system. The following sections present these three aspects of the framework extension.

5.1 White-Box Extensions

Subclassing `Entry` class serves two purposes: (1) to define application specific data fields and (2) to map the considered entry type to a specific configuration policy as shown in the assertions example in section 3.5.

The baseclass `Entry` doesn't have any predefined data fields. Subclasses define any kind of attributes: arrays like in LINDA, named instance variables like e.g. in JAVASPACES, or any mixed strategies.

Subclassing `SpaceAgent` class will be used for any application-specific adaption of space-using. E.g. summarizing several single accesses like the `append` method in the Market Place example in figure 3. Other examples that motivate specialization include handling of multiple space references or automatic subscription for notification.

Subclassing `OpenSpace` class allows one to introduce new or specialised operations like e.g. an update method which allows an agent to modify an entry at the space in one step, instead of having to read, modify and rewrite it. Or a direct exchange method, allowing to agents exchange their entries atomically, as realized in TSPACES with the `rhonda` operation. Many extensions however can be realized with suitably choosen configuration policies, as shown with the index handling example in 3.6.

5.2 Black-Box Extension: Configuration Policies

As we already mentioned, each space maintains a *mapping* between entry classes and their associated configuration policies. These policies determine the way the space controls accesses of entries. We discuss the detailed mechanism how this is organized. Then we present another variation of our market promoting an automatic garbage collection at the space.

Matching Algorithm The basic strategy for the associative lookup of entries at retrieving operations is simple: the space scans its entries collection for an entry that matches the given template. The ConfigurationPolicy provides the boolean function `does: anEntry matchWith: aTemplate` which is used for this scan (cf. form matching in figure 2). The state of the template may be controlled, special values may be required, a keyed matching may be performed. Any condition may be tested.

Pre- and Post-Access Hooks For each access operation, the `Configuration-Policy` class defines two *hook* methods. `preReadCheck: aTemplate` is called before the space's scan for a matching entry. It returns a validated version of the template that will be used for the lookup. To reject a template the method returns nil which causes the read operation method to abort and also return nil. After a successful lookup of a matching entry this is validated in the same manner by the `postReadCheck:` hook. This validated entry finally will be returned to the calling space agent. The take and write operations each employ an analogous pair of validators.

These hooks may be used for entry verification, modification, exchange or access refusal. Moreover, any additional actions can be triggered, like accessing the space to check for consistency, for doubles, perform some logging activity, etc.

5.3 Market Place V.5: Automatically Discarding Outdated Forms

Some automatic handling of outdated forms is a realistic requirement. Without such support, obsolete Requests and forgotten Offers can easily start to clutter a space. One possible solution for this is to add a timestamp to every entry being written to the space. After the expiry of the lifetime of the entry it will be discarded, this may be after a collectively defined duration for all entries or after an individually amount that is specified in the entry's fields.

The task of adding the arrival time at the space can be done by the `preWrite-Check:` method, by adding a key `#entryTime` with the current time to the form's bindings before writing it to the Space.

The check for expiration can be done periodically or triggered by the space accessing operations. Each of the hook methods of the configuration policy may call a garbage collection method that scans through all entries and discards the expired ones. This is sufficient for consistency since every access 'sees' a freshly updated view of the Space.

We implemented a Market Place variant with a default expiration time for all forms, using the policy's pre-operation hooks to trigger the cleanup. The following code extends the `FormPolicy` by creating a new subclass `LeasingPolicy` on which the

hooks methods are specialized. Figure 9 shows the specification of `preReadCheck:`. The boolean variable `isCleaning` is used to distinguish if the hook method is triggered by a client or by the configuration policy itself in the course of cleaning up.

5.4 Run-Time Configuration

Beside the static extension of the framework, OPENSPACES provides dynamic configuration of the spaces. Indeed, changing requirements of any kind can necessitate a reconfiguration of the used mapping. With OPENSPACES, an application can change the policies on a running system.

Dynamic Configurations. To define a configuration mapping, the space offers a registering method `register: anEntryClass withPolicy: #aConfigurationPolicy` that associates entry classes with specific policies. Since the needed configuration policy is looked up for every access of the space, a modification of the mapping automatically becomes effective. Therefore, it is easy to apply a new configuration on the fly. It is actually the standard procedure to register all needed entries after the creation of the space.

Note that the registration methods use the same mechanism for mutual exclusion as the basic operations. This guarantees that no running execution is interrupted.

Reconfigurations. There are situations that may require policies to be changed without restarting and resetting the entire system. New requirements or temporary changes may call for restrictions or modifications of the parameters.

To change the policy of an entry class we just unregister the old association before registering the new one. This is easily done with OPENSPACES. Dynamically changing the configuration mapping however can have an impact on the entries that are already present in the space, having been written under the previous policy. The OPENSPACES approach to this delicate problem is to give the programmer the possibility to specify necessary actions to be applied when a new policy is activated. To do so the space's method to register a new configuration calls a hook method of the new configuration policy, called `updateOldEntriesOfClass: anEntry`. It is executed after the new policies are activated, before any client may access the space thereafter. Any actions can be triggered for a clean transition. (It's the responsibility of the programmer to implement them!)

In Figure 10 we show the code used to dynamically introduce the lease time extension of the previous example. Any forms already present at the space should be supplied with an arrival time binding. The easiest way to do this is to take them all from the space and write them again, letting the new policy take care of it.

Caveats. The triggered actions in all policy hooks methods of the configuration policies may easily cause loops. When e.g. a `preReadCheck:` method of a configuration policy calls the read method with an entry of the same class as the policy is registered for, some precautions must be taken to prevent infinite loops. A solution can be to distinguish between the first call to the method (by the client) and the following calls (by the policy itself) with a flag that is set before the policy does its accesses and unset afterwards. Like that the hook is bypassed for 'internal use' (cf. figure 9).

```
FormPolicy subclass: #LeasingPolicy
  instance variables: 'defaultLeasingTime isCleaning'.

LeasingPolicy >> preWriteCheck: aTemplate
  |checkedTemplate|
  checkedTemplate := super preWriteCheck: aTemplate.
  checkedTemplate notNil
    ifTrue: [aForm bindings at: #arrivingTime put: Time now].
  ^checkedTemplate

LeasingPolicy >> preReadCheck: aTemplate
  |checkedTemplate|
  ^ self isCleaning   "Is the method initiated by the receiver?"
    ifTrue:
      [aTemplate]     "Do not cause a loop"
    ifFalse:
      [checkedTemplate := super preReadCheck: aTemplate.
      self cleanupSectionOfTemplate: checkedTemplate]

LeasingPolicy >> preTakeCheck: aTemplate
  "... same as preReadCheck: "

LeasingPolicy >> cleanupSectionOfTemplate: aTemplate
  "Enter 'cleaning-mode', perform cleanup and exit again"
  |section|
  self isCleaning: true.
  section := aTemplate bindings at: #section.
  self throwAwayOutdatedFormsAtSection: section.
  self isCleaning: false.
  ^aTemplate

LeasingPolicy >> throwAwayOutdatedFormsAtSection: aSectionName
  |template forms arrived|
  template := Form new.
  template bindings at: #section put: aSectionName.
  forms := hostSpace readAll: template.
  forms do: [:each |
    (each bindings includesKey: #arrivingTime)
      ifTrue:
        [arrived := each bindings at: #arrivingTime.
        ((arrived addTime: defaultLeasingTime) < Time now)
          ifFalse: [hostSpace take: each]]
```

Fig. 9. The configuration policy for automatic garbage collection.

```
LeasingPolicy >> updateOldEntriesOfClass: anEntryClass
  "Take all present forms from the space and rewrite them. Like
  that they are supplied by the policy with a binding denoting
  they had just arrived."
  | template forms |
  template := Form new.
  forms := hostSpace readAll: template.
  forms
    do: [:each | hostSpace write: each]  "adds arriving time"
```

Fig. 10. The update method for the dynamic introduction of the lease time policy.

It is also not recommendable to use the blocking variants of the retrieving operations because they would cause the main space process to block.

6 Related work

There are several parameters regarding configurability of a data-space framework: is it object-oriented, does it support white-box extensions, are there options for pluggable configurations in the black-box style? We will focus on three implementations promoting configurability options.

R.Tolksdorf's BERLINDA [Tol97] is a object-oriented framework similar to OPENSPACES. It has a set of basic abstractions for a space, entries, agents. It can be extended in a white-box style. In BERLINDA, a concrete entry class has to implement the matching function, which can be defined as desired. Like in OPENSPACES, this feature is remarkable, since all other known implementations use a fixed algorithm, which cannot be modified. Note however that in OPENSPACES the matching algorithm is decoupled from the entries and then can be changed dynamically.

Concerning access control, the work of Minsky and Leichter [ML95] on LAW-GOVERNED LINDA is of great interest. In this model a *Law* rules the reactions of the tuple space to events occurring on attempts to use the access operations or when a successful matching has been performed. These events may trigger actions that are defined in the global law. [MMU00] introduces controller-processes for each agent using a space which enforce the application of the law.

OPENSPACES has a similar reaction model. The events are all types of access operations at the space, the reactions are the different hooks methods that are applied before and after each of them. By modifying or exchanging a used template or a found entry in the hook methods we can model LAW-GOVERNED LINDA's variants of operation completions. The two different post-retrieving hooks allow one additionally to distinguish between a matching of a read or of a take operation. The enforcing of the rules in OPENSPACES is encapsulated in the configuration policies of which are local to the space. This has the advantage that a reconfiguration does not have to modify multiple copies of a law that are spread amongst the controllers. Moreover, OPENSPACES does not restrict the space operations that may be used as reactions, but it is left to the framework user to be aware of potentially blocking methods or loops.

In “Programmable Coordination Media” Denti et al. show well the principal benefits of a space with programmable behaviour [DNO97]. Indeed, their specialization of the space behaviour to enforce resource accessing strategies inspired us for searching ways to let a space take care of additional responsibilities. This helps to free the agents from unnecessary responsibilities and adds control to the space.

Two popular tuple space frameworks in Java have been released in ’99: Sun Soft’s JAVASPACES [FHA99] and IBM’s TSPACES [WMLF99]. Both use subclassing of a general entry (resp. tuple) abstraction for the exchanged data. The matching algorithm is fixed for both of the systems. TSPACES provides some interesting extensions like the mentioned `rhonda` operator, or a set of query-like range matching options and set-retrieval operations.

In summary, while LAW-GOVERNED LINDA offers a fine grained access control at the space, most implementations provide at most extension through subclassing. The matching algorithm is definable in BERLINDA, in all other works it is fixed. Moreover, dynamic reconfiguration of the space behaviour is not addressed.

7 Conclusion

In this paper we have presented OPENSPACES, an object-oriented framework for building applications with architectures in a blackboard style. The kernel of OPENSPACES is based on six entities: *Space agents* access *spaces* to store or retrieve *entries*. *Configuration policy* objects are responsible for the matching strategy and for a complete set of access controlling methods. The *space server* provides access for the space agents and the *space administrator* manages creation and lifecycle of the spaces.

We have presented several extensions to show how the design of the framework allows the developer to control all the important parameters of space manipulations. Beside *white-box extension* based on the specialization of the core entities, OPENSPACES allows us to extend it in a *black-box* style by defining new policy objects that can be plugged to configure the space’s behavior. This reconfiguration may be done *dynamically* during runtime. OPENSPACES offers support to handle the transition between two consecutive configurations.

For the future we plan to extend our prototypical framework with support for pure CORBA using IDL. This will allow clients agent written in other languages to use OPENSPACES for coordination.

Acknowledgements The authors would like to thank Juan-Carlos Cruz for his constant and competent support.

References

[CR96] P. Ciancarini and D. Rossi. Jada: Coordination and communication for java agents. In *MOS’96: Towards the Programmable Internet*, LNCS 1222, pp. 213–228, Linz, Austria, July 1996. Springer-Verlag.

[DNO97] E. Denti, A. Natali, and A. Omicini. Programmable coordination medium. In D. Garlan and D. Le Métayer, editors, *Proceedings of COORDINATION’97 (Coordination Languages and Models*, LNCS 1282, pp. 274–288. Springer-Verlag, 1997.

[FHA99] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces Principles, Patterns and Practice*. Addison-Wesley, 1999. ISBN: 0201309556.

[GC85] D. Gelernter and N. Carriero. Generative communication in linda. *ACM TOPLAS*, 7(1), January 1985.

[JF88] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[ML95] N. H. Minsky and J. Leichter. Law-governed linda as a coordination model. In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, LNCS 924, pp. 125–146. Springer-Verlag, 1995.

[MMU00] N. H. Minsky, Y. M. Minksy, and V. Ungureanu. Making tuple space safe for heterogeneous distributed systems. In *Proceedings of SAC'2000*, pp. 218–226, 2000.

[PA98] G. A. Papadopoulos and F. Arbab. Coordination Models and Languages. In *The Engineering of Large Systems*, volume 46 of *Advances in Computers*. Academic Press, August 1998.

[Tol97] R. Tolksdorf. Berlinda: An object-oriented platform for implementing coordination languages in java. In *Proceedings of COORDINATION'97 (Coordination Languages and Models*, LNCS 1282, pp. 430–433. Springer-Verlag, 1997.

[WMLF99] P. Wyckoff, S.W. McLaughry, T.J. Lehman, and D.A. Ford. T spaces. *IBM Systems Journal*, 37(3), 1999.