The Knowledge Engineering Review, Vol. 24:1, 77–93. © 2009, Cambridge University Press
doi:10.1017/S0269888909000149 Printed in the United Kingdom

Engineering spatial concepts

DANIELA MICUCCI, FRANCESCO TISATO and MARZIA ADORNI

D.I.S.Co., University of Milano—Bicocca, viale Sarca 336, 20126 Milan, Italy;
e-mail: daniela.micucci@unimib.it, francesco.tisato@unimib.it, marzia.adorni@gmail.com

Abstract

The success of a software system strongly depends on the ability of turning a precise domain analysis into a concrete architecture. Even if the domain model relies on sound ontological bases, there is often a wide semantic gap between the conceptual model and the concrete components that should reify it. To fill the semantic gap, relevant domain concepts should be engineered by identifying the corresponding architectural abstractions, which can be realized by concrete software components. Space plays a crucial role in many application domains, but surprisingly, related architectural abstractions have not emerged yet. This paper proposes space-related abstractions derived from the application of classical software engineering principles; in particular, the information hiding principle that leads to an operational definition of space. Basic abstractions are refined to deal with architectural aspects. As the underlying software engineering principles are close to principles that underlie the definition of space ontologies, the conjecture is that the proposed space architectural abstractions might be the basis for a formalization in ontological terms.

1 Introduction

Crucial aspects in the development of a complex IT system are the identification of both a sound domain model, possibly derived from an ontology-based approach, and a sound architectural model.

Software architecture is still a growing discipline. Hence, it has no single and accepted definition (Bass *et al*., 2003). Over the years, many definitions have been formulated: comprehensive reviews may be found in Bass *et al*. (2003) and in the Software Engineering Institute, Carnegie Mellon (2008). For our purpose, we may refer to the following one: software architecture 'refers to the structure of the system quite hidden from the user' (Clements, 2008). Then it captures relevant issues that are loosely related to domain issues. This results in a wide semantic gap between domain models and architectural models. Even if the model of an application domain relies on sound ontological bases, its mapping onto an architecture that effectively reifies the model is a difficult task.

To fill the semantic gap, relevant domain concepts should directly turn into architectural abstractions, that is, architectural patterns and components. The abstractions should be identified according to well-established software engineering principles.

The concept of *space* plays a crucial role in many application domains. Surprisingly, in the software engineering area, space-related architectural abstractions have not emerged yet, even though philosophers and scientists have dealt with spatial ontologies since millenniums. In real-life software systems, space-related aspects are usually intermixed with domain-specific and implementation-dependent aspects. The identification of orthogonal space-related abstractions is still an open issue, the solution of which should rely on the conceptual framework provided by spatial ontologies (Bateman & Farrar, 2004b) and by research achievements in several application domains: in particular, the geographical information system (GIS) (Perry *et al*., 2006), surveillance (Howarth, 2005) and robotics (Bateman & Farrar, 2004a).

Usually a software architect identifies architectural abstractions by relying on 'architectural commonsense', that is, by exploiting the principles of software engineering as separation of concerns, abstraction and encapsulation. Such principles should be applied to the identification of spatial architectural abstractions too.

On the other hand, it can be observed that concepts close to architectural commonsense underlie the definition of space ontologies. Consequently, our conjecture is that spatial architectural abstractions may provide useful guidelines for the definition of related ontologies.

The objective of this paper is to contribute to the cross-fertilization between ontologies and software architectures by focussing on the concept of space. Some preliminary ideas have been presented in Tisato *et al.* (2007).

Section 2 reviews established software engineering concepts and their relationships with relevant issues in the spatial ontologies area. Section 3 introduces a simple example that will be exploited throughout the rest of the paper. Section 4 presents basic architectural abstractions that could help in building space-aware systems; the conjecture is that such abstractions are reasonable candidates for a formalization in ontological terms. Section 5 extends the basic abstractions towards architectural aspects. Finally, Section 6 draws some conclusions and tries to stimulate the discussion between different research communities.

2 Space awareness and software engineering

This section reviews a few established software engineering principles and their relationships with emerging issues in the spatial ontologies area. The aim is to highlight the similarity of concepts used in the ontologies and in the software architectures areas. Comprehensive reviews of spatial ontologies and of software engineering principles can be found in Bateman and Farrar (2004b) and Ghezzi *et al.* (2003), respectively.

Separation of concerns (Parnas, 1972) refers to the ability to identify, encapsulate and manipulate only those parts of software that are relevant to a particular concept, goal or purpose. Concerns are the primary motivation for organizing and decomposing software into manageable and comprehensible parts (Tarr *et al.*, 2000). Applying separation of concerns to the concept of space means, for instance, that the intrinsic features of a car or a person should not depend on their locations. On the contrary, it is useful to know whether there is an entity at a given location, no matter whether it is a car or a person.

Separation of concerns has been recognized as a key issue by the spatial ontologies community. As pointed out in Pelekis *et al.* (2004), both theoretical and technical frameworks are lacking, which explicitly describes the fundamental spatio-temporal characteristics in a context-independent manner. Yuan (1996) introduces the three-domain model, which carefully separates theme, space and time. Parent *et al.* (1999) highlight the need for a clear separation between entities and locations. Bateman and Farrar (2004b) outline that equating objects with their locations is at least problematic, and that the particular location of an object is not a necessary ontological feature of that object. Perry *et al.* (2006) exploit relationships in the thematic domain to indirectly associate spatial properties with entities.

Encapsulation is a basic technique to ensure separation of concerns. Encapsulation means that the focus is not on *what* an object is, but on *how* it behaves from the point of view of an external agent—be it a human being or a software component. Object-orientation reifies this concept by separating the interface of a class from its implementation. In layered systems, a higher layer exploits functionalities exported by a lower layer without knowing its internal structure. In service-oriented architectures, the requestor of a service is not aware of how it is implemented.

Several ontological approaches endorse the concept of encapsulation. The focus is not on *what* a space is, but on *how* we can reason about a space; for example, by defining the set of performable queries about locations. This approach is consistent with the database-oriented attitude of the GIS community; Perry *et al.* (2006) define a comprehensive set of operators on spatial databases. Other approaches focus on how an object can move around the space. In Egenhofer and Rodrguez

(1999), spatial relations are derived from schemes of action. In Bateman and Farrar (2004c) 'the structure of space…is measured by how we can move in it and how objects can be placed within it', according to the Gibsonian concept of affordance (Gibson, 1977).

The capability of viewing and organizing entities under different *perspectives* is another aspect of separation of concerns. This is a widespread concept in relational and multi-dimensional databases and, in general, in software engineering.

Complex systems require *several space models* to co-exist. Possibly different models correspond to different levels of abstraction. For example, a high-level spatial model of a building can be a graph where each node is associated with a room, and an edge is associated with a passage; this level of abstraction may support the strategic planning of a robot motion as a path over the graph. On the other hand, the fine-grained trajectory of the robot is defined in a Euclidean space. The same entity (a robot) must be located in both spaces: once the high-level path of the robot has been planned in terms of a sequence of nodes, it must be mapped onto Euclidean coordinates to define the fine-grained trajectory. Therefore, there is the need for a *mapping* between the two spaces.

Often the concept of space must be extended to model conceptual spaces and their mappings. For example, the localization of an employee office may require a mapping between an organizational space (the company organization chart) and a graph space (the company offices).

Significant research activities on ontologies deal with multiple perspectives and, in particular, with multiple spaces. The realist perspectivalist approach (Smith & Grenon, 2004) embraces not only one ontology but a multiplicity of complementary ontologies—distinct perspectives of reality, each one of which is veridical. Howarth (2005) discusses different kinds of spaces in video-surveillance systems, mappings between spaces, and knowledge representation that is to be used by another computer programme. Kuipers (2000) shows how multiple interacting, qualitative and quantitative, topological and metrical and spatial representations can be integrated in a common framework, that can express states of partial knowledge. Bateman and Farrar (2004b) highlight the fact that there may not be a single ontological perspective that is sufficient for all tasks; hence, a foundational ontology needs to provide for the different perspectives that can be taken regarding phenomena in the world. Within GIS, a range of perspectives have always been required, and a major issue is that of reconciling these perspectives and providing mappings across them. There are substantially different ontological domains that need to be maintained separately; re-use and modularization are inevitable for an effective treatment. There are several proposals of stratified and tiered ontologies, in which various ontological perspectives may be organized and related; see, for instance, Borgo *et al.* (1996) and Frank (2001). Fonseca *et al.* (2002) exploit the concept of levels of abstraction and suggest the use of semantic translators as a powerful solution for interoperability.

3 An example

Figure 1 shows the plant of a building (for example, a warehouse). Black boxes model obstacles. Several agents, be they human beings or software components, cooperate to perform a task that, from the spatial point of view, turns into moving a vehicle from one location to another. Agents play four major roles. A *supervisor* defines the task on the basis of domain thematic concepts: for example, a vehicle in Room A must go to Room B to collect some goods. A *coarse grain planner* defines a coarse-grained path aimed at minimizing the number of traversed rooms. A *medium grain planner* defines a medium-grained path aimed at avoiding obstacles and at minimizing the number of movements the vehicle has to perform inside each traversed room. Finally, a *controller* controls the fine-grained movements of the vehicle.

The supervisor only deals with domain thematic concepts, whereas the other agents rely on three spatial representations of the building, corresponding to different spatial models at different levels of abstraction (Figure 2):

- a *graph model*, in which nodes correspond to rooms and edges correspond to passages between rooms. The coarse grain planner exploits this model to define the coarse-grained path;

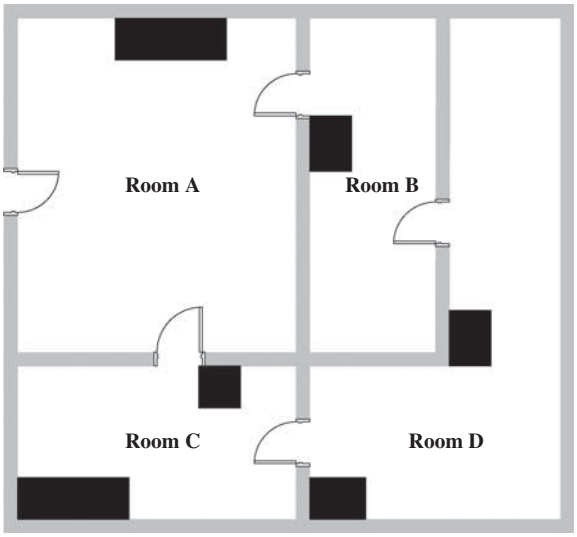

Figure 1 The plant of the building

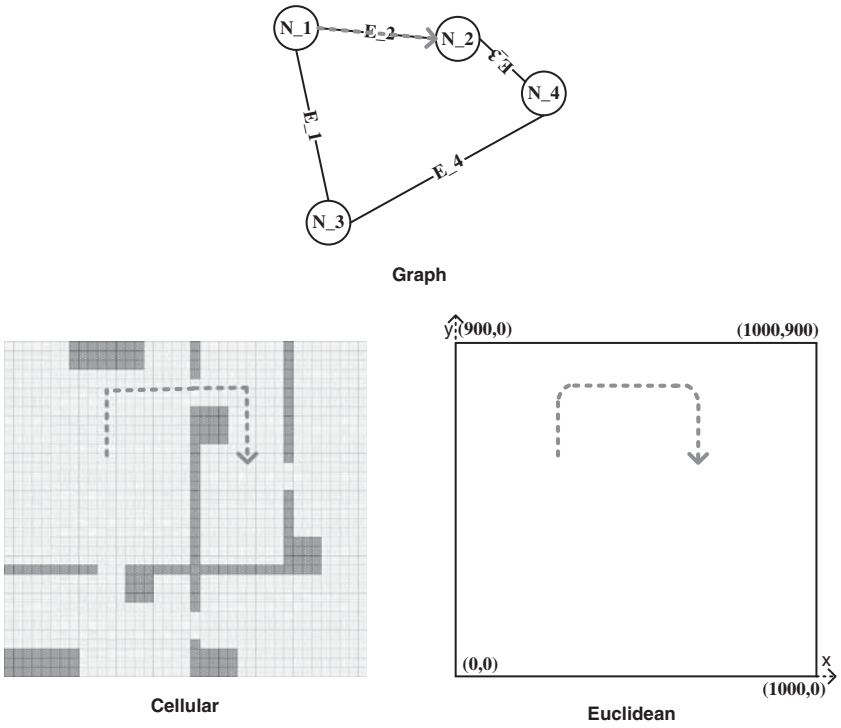

Figure 2 Spatial representations of the building

- a *cell model*, in which cells are small rectangular areas that can be either traversable or not (walls and obstacles). The medium grain planner, similar to many robotic systems (Marchese, 2005), exploits this model to efficiently compute a medium-grained path;
- a *Euclidean 2D model*. The controller exploits this model to compute and control the continuous movements of the vehicle.

More precisely, if the vehicle is somewhere in Room A and the task implies moving to Room B, the graph model is exploited to state that the less-costly coarse-grain path is a direct movement from A to B. The cell model is exploited to define a medium-grain path as a sequence of movements

between adjacent locations (i.e., cells), which leads from the current location of the vehicle to a cell belonging to B (for e.g., its barycentre). The Cartesian model is exploited to define a fine-grain path as a sequence of movements between locations (i.e., 2D coordinates), which smoothens the sequence of segments connecting the centres of the cells.

This simple example suggests some basic remarks:

- Reasoning at all levels relies on the concepts of *location*, that is, 'somewhere' in a space.
- A location is defined by specifying a *movement* from a previous one.
- It must be possible to associate a *cost* to a movement.
- The manner in which movements are specified characterizes a *space model*. In the example, movements for the three space models are specified by graph edges, directions of moving between adjacent cells and pairs of real numbers, respectively.
- The overall process implies a *mapping* between locations defined in different spaces: from a node (i.e., room) to a cell corresponding to its barycentre, from a cell to a Cartesian coordinate, and vice versa.

The basic operations above do not depend on thematic domain concepts, for example, why the vehicle moves, what it does once the destination has been reached, which criteria should be used to optimize the paths and so on. According to the principle of separation of concerns, thematic and space-related aspects should be kept carefully separated from both an ontological and a technological point of view.

These basic operations are more precisely defined in Section 4. Additional concepts, dealing with the overall architecture of a space-aware system, are presented in Section 5.

4 Abstractions

This section introduces a basic set of space-related abstractions. The key issue is that, according to the encapsulation principle, a space is characterized by the operations an agent can perform. In computer science parlance, a space is viewed through an *operational interface*. The interface encapsulates the intrinsic spatial structure, which, from the agent's point of view, is a metaphysical issue (or, in the case of a software application, a hidden implementation issue).

The architectural abstractions are described in Unified Modelling Language (UML)-like style. Though a discussion on UML is out of the scope of this paper, it should be pointed out that the formal definition of UML2 (OMG, 2008) is sound enough to consider it as a suitable formalism for the definition of spatial ontologies.

4.1 Generic space

The `GenericSpace` interface (Figure 3) models a generic space by specifying a set of performable operations. The interface is generic with respect to the type of movements that appear in the methods signature. This is the only issue from an agent's point of view, which differentiates space models.

The interface (therefore any of its implementations) depends on three basic concepts modelled by `Location`, `Cost` and `Movement`.

An agent may ask a space for a location through the `defLocation()` operation by specifying a starting location and a movement. The returned location is an opaque handle, that is, its internal representation cannot be inspected or manipulated; therefore the `Location` class does not expose public operations. From this point of view there is a similarity with regard to the concept of reference in the Java programming language: you can get a reference to something, but you can neither inspect nor manipulate the internal representation of the reference.

The definition of a location is a pure functional operation: a space creates locations, but it neither keeps track of them nor do they affect its state. Once a location has been created and returned to a requesting agent, the lifetime of the locations is entirely up to the agent. This is modelled by the directionality of the association from `Location` to `GenericSpace` and by the

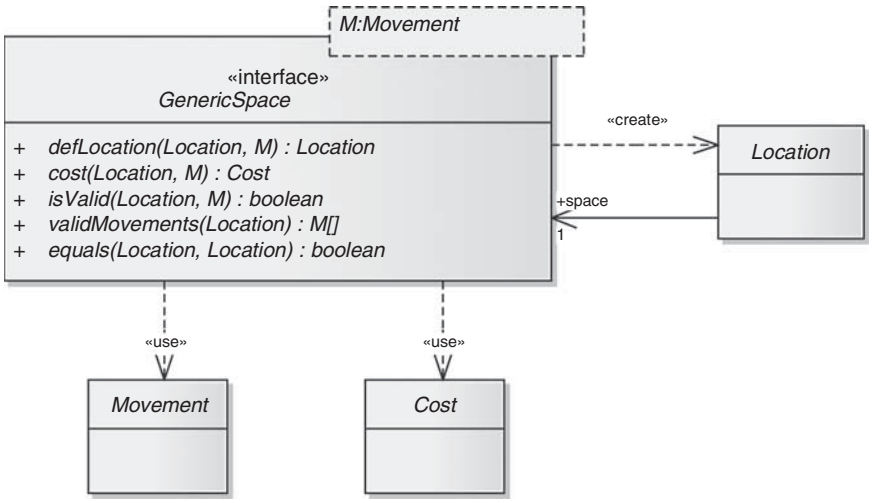

Figure 3 Generic space

<<create>> dependency in Figure 3: a location is meaningful in the context of a space, but a space is not aware of the locations it defines.

The `cost()` operation returns the cost of a movement from a location. For instance, in a Euclidean space, a cost may be the distance between two locations; in a graph space, a cost may be the weight of an edge. Given a location, `L`, and a movement, `M`, a `cost(L,M)` call returns a measure of the effort needed to perform the movement `M` from location `L`.

The `isValid()` operation states whether a movement from a location is valid. For example, a movement in a Cartesian space the target location of which falls out of the boundaries of the space (see Subsection 5.2), is not valid.

The `validMovements()` operation returns the set of valid movements from a location. Though it may be useful in several cases (for instance, for a graph space), it is meaningless for continuous spaces (for instance, a Euclidean space). This issue is a subject for future investigation.

The `equals()` operation states whether two locations can be considered coincidental. The concept of equality depends on the typology of the space model. In a discrete space (for e.g., a graph or a cell space) it is obvious. In a continuous space (for e.g., a Euclidean space) it turns into the concept of closeness.

The above operations are the basic ones and they are the only operations that are parametric with respect to movement type. Other operations will be discussed in the following.

4.2 Space models

The generic space interface is realized by specific *space models*, such as a graph model, a cell model, a Euclidean2D model, and so on (Figure 4, in which only one typology of space model is detailed). As pointed out before, space models are parameterized by the type of performable movements. Therefore, a model realizes the `GenericSpace` interface by specifying a movement type.

The `Movement` class is specialized according to space models. In `Euclidean2DSpace`, a movement is modelled by `2DMov`, that is, a couple of long values; in `CellSpace` a movement is modelled by `CellMov`, that is, a value within an enumeration type (e.g., NORTH, SOUTH, EAST, WEST); in `GraphSpace` a movement is defined by `GraphMov`, that is, the symbolic identifier of an edge.

The implementations of the operations of `GenericSpace` must preserve the semantics of each space model. For example, the `defLocation()` operation in `Euclidean2DSpace` must ensure that parallel movements from different locations do not lead to equal locations, and the `cost()` operation must ensure that movement costs (i.e., distances) respect the Pythagoras theorem.

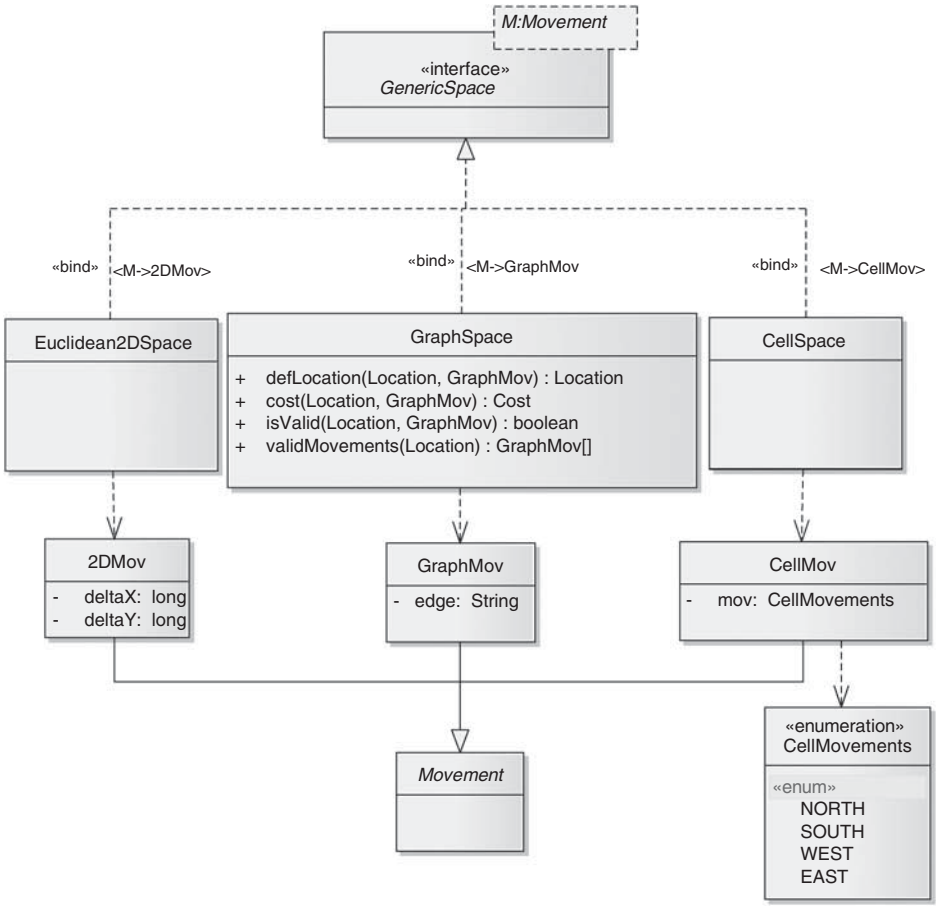

Figure 4 Space models

4.3 Mapping

Agents reason in terms of locations defined by different spaces that, in turn, rely on different space models. In many cases, as in Figure 2, spaces are related as they represent the same conceptual entity under different perspectives, example, the building. Therefore, a *mapping* must be established between locations defined in different spaces, example, between a graph node representing a room and a cell representing its barycentre. The `SpaceMapping` class (Figure 5) embodies the mapping process. It is associated to one source space and to one target space.

The `getMapped()` method, given a location in the source space, returns a possibly empty set of locations in the target space.

`GraphToCellMapping` (Figure 5) exemplifies a mapping between locations in a graph space and locations in a cell space, which will be exploited in the examples. For simplicity, the `getMapped()` operation is assumed to return one cell location only, example, the location in the cell space of the barycentre of the room corresponding to a node in the graph space.

The returned set may be empty, for example, the cell space does not represent the whole building. On the other hand, the returned set may include multiple locations, for example, it may include all the cells of the room modelled by a node.

Implementations of `SpaceMapping` may depend on the internal implementation of the space model, which will be discussed in Subsection 5.2.

4.4 Example

Figure 6 sketches a concise communication diagram highlighting how agents cooperate in the example problem. Each agent relies on a single space model and exploits a specific space

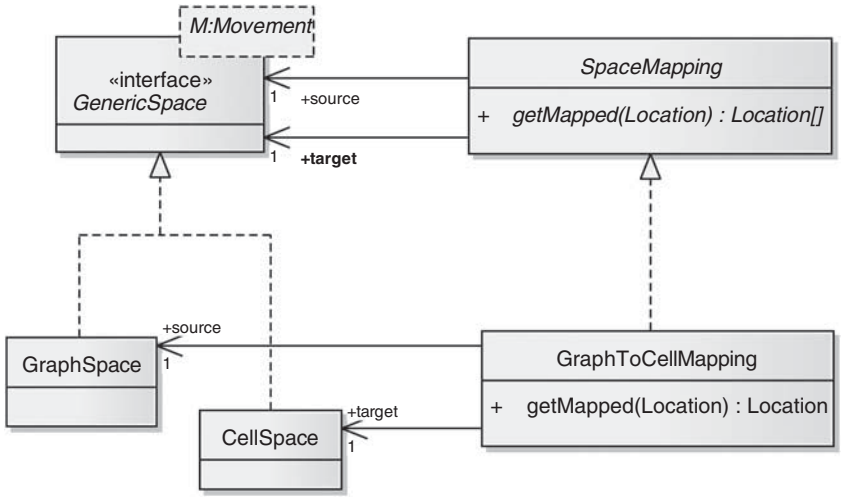

Figure 5 Space mapping

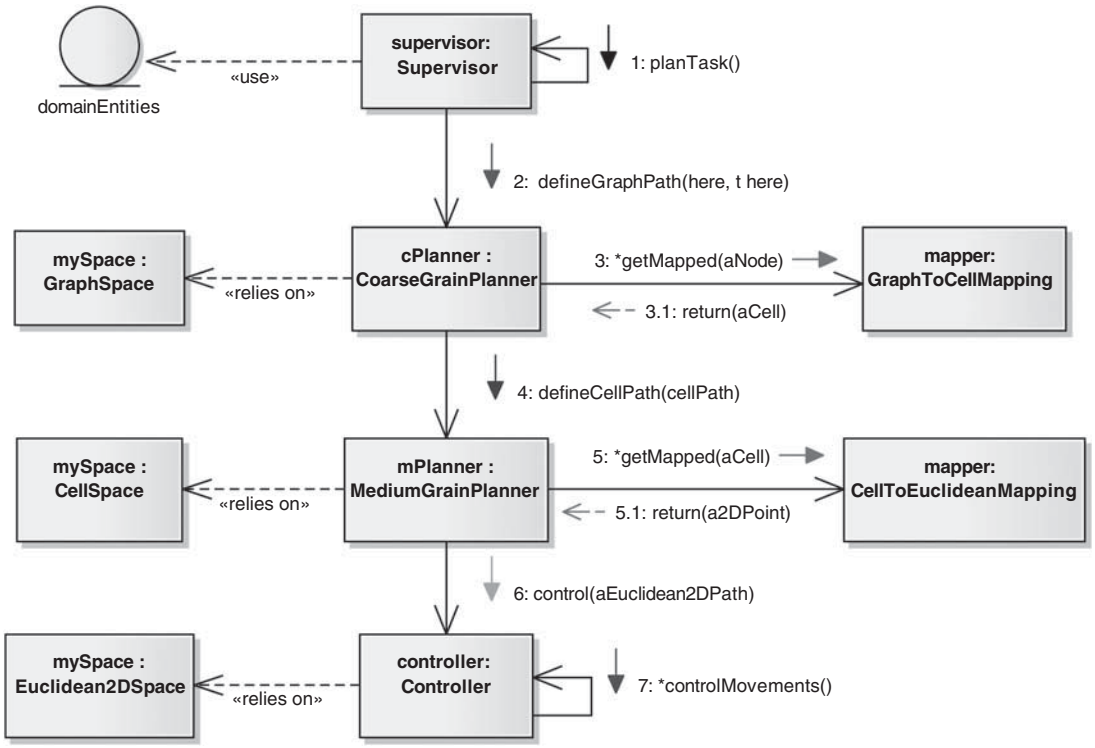

Figure 6 Agents cooperation

mapping to convert locations defined in its own space into locations in the space of the lower level agent.

The snippets of pseudo-code in Figures 7 and 8 detail the communication diagram in Figure 6 by highlighting how agents use the operations exposed by the spaces they rely on. The supervisor (Figure 7) reasons in terms of thematic domain entities, example, Room, which model space-independent concepts. Rooms are also associated to a location in a graph space. Once the supervisor decides that a vehicle should move from Room A to Room B to collect some goods, it notifies the coarse grain planner that a path between the locations of Rooms A and B must be defined.

```
public class Supervisor {

    //thematic domain information
    class Room {
        String name;                //symbolic name of the room
        Goods[] goods;              //goods stored in the room
        Location location;          //room location in the graph space
    }

    CoarseGrainPlanner cPlanner;    //the exploited coarse grain planner

    Room roomA, roomB;              //thematic domain entities

    //(initialization details omitted, see Section 5)

    //define the vehicle task
    public void planTask(){
        //...
        //decide that the vehicle must go from Room A to Room B...
        //...ask the planner to define a path between A and B locations
        cPlanner.defineGraphPath(roomA.location, roomB.location);
        //...
    }
}
```

Figure 7 A supervisor

```
public class CoarseGrainPlanner {

    GraphSpace mySpace;             //the graph space exploited by planner

    GraphToCellMapping mapper;      //the SpaceMapper

    MediumGrainPlanner mPlanner;    //the exploited medium grain planner

    Location[] graphPath;           //the path in terms of graph locations

    Location[] cellPath;            //the path in terms of cell locations

    //(initialization details omitted, see Section 5)

    //the planner operation invoked by a Supervisor
    public void defineGraphPath(Location here, Location there) {

        //a naive algorithm for defining a path

        //get the valid movements from location "here"
        GraphMov[] validMov = mySpace.validMovements(here);

        //for all valid movements aMov from here... {

            //get the cost of the movement
            Cost aCost = mySpace.cost(here, aMov);

            //get the target location of the movement
            Location loc = mySpace.defLocation(here, aMov);

            //...add loc to graphPath...
            //...evaluate the overall cost of the path...

            //final location reached with minimal cost?
            if ((loc.equals(there) and ("cost OK")) exit;
            else {...} //...recursively repeat the process
        }

        //map graph locations into cell locations
        //for all locations aLocation in graphPath...
        cellPath.add(mapper.getMapped(aLocation));

        //pass the cell path to the medium grain planner
        mPlanner.defineCellPath(cellPath);
    }
}
```

Figure 8 A coarse grain planner

The coarse grain planner (Figure 8) reasons in terms of graph location to identify a path between the initial `here` location and the final `there` location. The example is aimed at highlighting how the agents exploit the space operations; therefore, the algorithm used for finding an optimal path is not relevant for the discussion. In the end, the planner exploits the `getMapped()` operation to translate graph locations into cell locations and passes the resulting medium-grained cell path to the medium grain planner.

Both medium grain planner and controller are not shown for brevity. The medium grain planner relies on cell movements to refine the cell path by exploiting the `validMovements()` operation on the cell space. Then it maps cell locations onto Cartesian locations and passes them

on to the controller. The controller relies on 2D movements to smoothen the Euclidean 2D path and to control the physical movements of the vehicle.

This simple example highlights several relevant issues.

Domain thematic aspects are carefully separated from space-related aspects. The former ones are encapsulated inside the domain-aware supervisor, whereas the latter ones are encapsulated by the space-aware planners and controller.

A space-aware agent reasons at the proper level of abstraction in terms of a single space model and is in charge of mapping spatial information into the lower level model. An agent is not aware of higher-level space models.

Space-aware agents are not aware of the concrete structure of the spaces they rely on, as they exploit abstract operations on locations and movements.

Ultimately, the overall structure is well layered by improving separation of concerns and encapsulation.

There is at least one defect in the example: the supervisor is somehow space-aware, as it is in charge of maintaining the association between domain entities (e.g., rooms) and their locations. This is modelled by the `location` attribute in the `Room` class inside `Supervisor` (see Figure 7). Therefore, there is still some intermixing of domain and spatial issues.

Moreover, the above example skips a major issue, that is, how to define and initialize the concrete layout of a space. These problems are tackled in the next section.

5 Architecture

Section 4 introduced basic spatial abstractions in terms of operations exposed by interfaces. This section introduces additional concepts that are more related to architectural issues: how to support a symbolic naming for space-related entities, how to define and initialize the layout of concrete spaces and how to ensure the harmonious cooperation among agents, which rely on a subjective vision of the space.

5.1 Markers

As pointed out in Subsection 4.4, the use of the basic abstractions alone implies that an agent who deals with the thematic application domain is in charge of maintaining associations between domain entities and locations. Though it works, it does not properly model real-life situations, in which locations are associated with symbolic *markers*. The association may be persistent, as for town names on a map. They can be transient and defined by agents, such as markers a user puts at a location on a Google map. Finally, they can be mobile, such as a marker denoting the current location of a vehicle. Symbolic markers support the full separation between thematic and spatial concepts, because domain-related reasoning may rely on symbolic identifiers only.

A `Marker` (Figure 9) is a persistent object, which embodies a symbolic identifier (e.g., 'ROOM_A') plus an association with a location. A `MarkedSpace` is an implementation of `GenericSpace`. It hosts a collection of persistent markers, the symbolic names of which may have a semantic interpretation in the thematic domain, and it implements a mapping from markers to locations. Ultimately, a `MarkedSpace` plays the role of a name server that maps markers onto locations.

Note that `MarkedSpace` is not a pure interface, as it maintains a collection of persistent markers. However, `MarkedSpace` is still abstract, as it is realized by concrete spaces. For example, `GraphSpace` and the other concrete spaces in Figure 4 may realize `MarkedSpace` in order to provide a naming service. The detailed class diagram is straightforward and is omitted for brevity.

The `defMarker()` operation allows a marker with a given symbolic id to be associated with a location.

The `locationOf()` operation returns a location, given a symbolic marker id.

The `markersAt()` operation returns the symbolic ids of all the markers, if any, at a given location (or reasonably close to the location in case of a continuous space).

The `moveMarker()` operation moves a marker from one location to another.

The `allMarkers()` operation returns the ids of all the markers defined in the space.

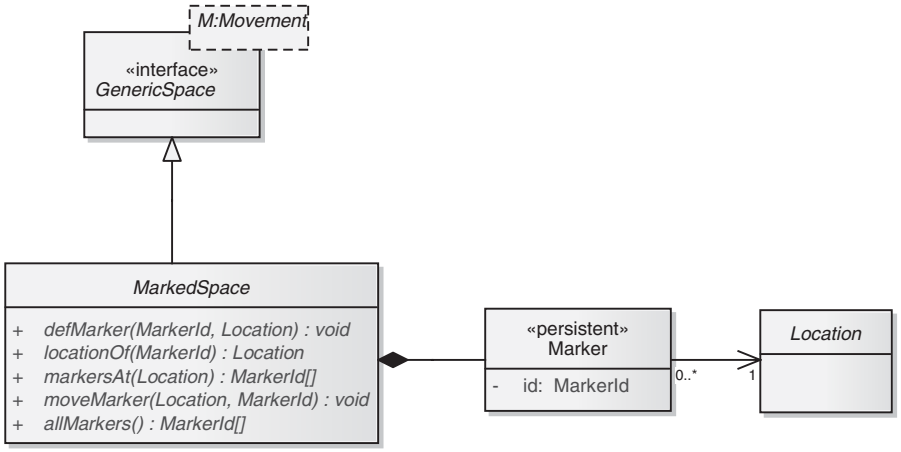

Figure 9 Markers

5.2 Layout

A concrete space is an instance of a space model, initialized with a *space layout* as denoted by constructors in Figure 10. A space layout is an instance of the `Layout` class that provides a concrete description of the specific features of a space, defined according to its model.

Figure 10 exemplifies some possible implementations of `Layout`. A graph layout may be implemented as a collection of objects modelling nodes and edges. A cell layout may be implemented as an array of cells that can be either traversable or not. A Euclidean 2D layout, which implements a continuous space with a potentially unlimited number of locations, only specifies some constraints, example, boundaries and measurement unit.

To be general, specific layouts should be defined in turn as abstract classes, which may have different implementations; the only constraint is that the implementation of a layout must properly support the implementation of the operations of the space model. For example, a graph layout could be implemented both as a collection of linked objects representing nodes and edges, and as a matrix where rows and columns represent nodes, and cells represent the presence of edges. Of course, subclasses of `Layout` must provide implementation-dependent methods for defining the concrete layout of a space.

As pointed out in Subsection 4.3, `SpaceMapping` implementations also depend on layout implementations. In particular, an implementation of `SpaceMapping` is expected to provide a `setMapping()` method for establishing the mapping among locations in the source and target space. In some cases, the mapping must be explicitly specified: for example, the mapping from graph nodes to cells. In other cases, it may be implicit: for example, the mapping from regular cells to the Cartesian coordinates of their barycentre.

5.3 Well-known markers

The definition of concrete layouts is up to implementation-aware agents, as we shall discuss later. Most agents are space-aware, not implementation-aware: they reason in terms of opaque locations and movements only by exploiting space operations exposed by space models, which encapsulate concrete layout implementations.

A space-aware agent must know at least one initial position to start reasoning about space in terms of movements. Implementations of the abstract `anchor()` method of `Layout` allow an implementation-aware agent to define *well-known markers*, which are anchored to the concrete implementations of locations. As sketched in Figure 10, the parameters of `anchor()` implementations depend on layout implementations.

Symbolic identifiers of the well-known markers are published by the space; this is a major usage of the `allMarkers()` operation of `MarkedSpace` (Figure 9). For example, a Euclidean space

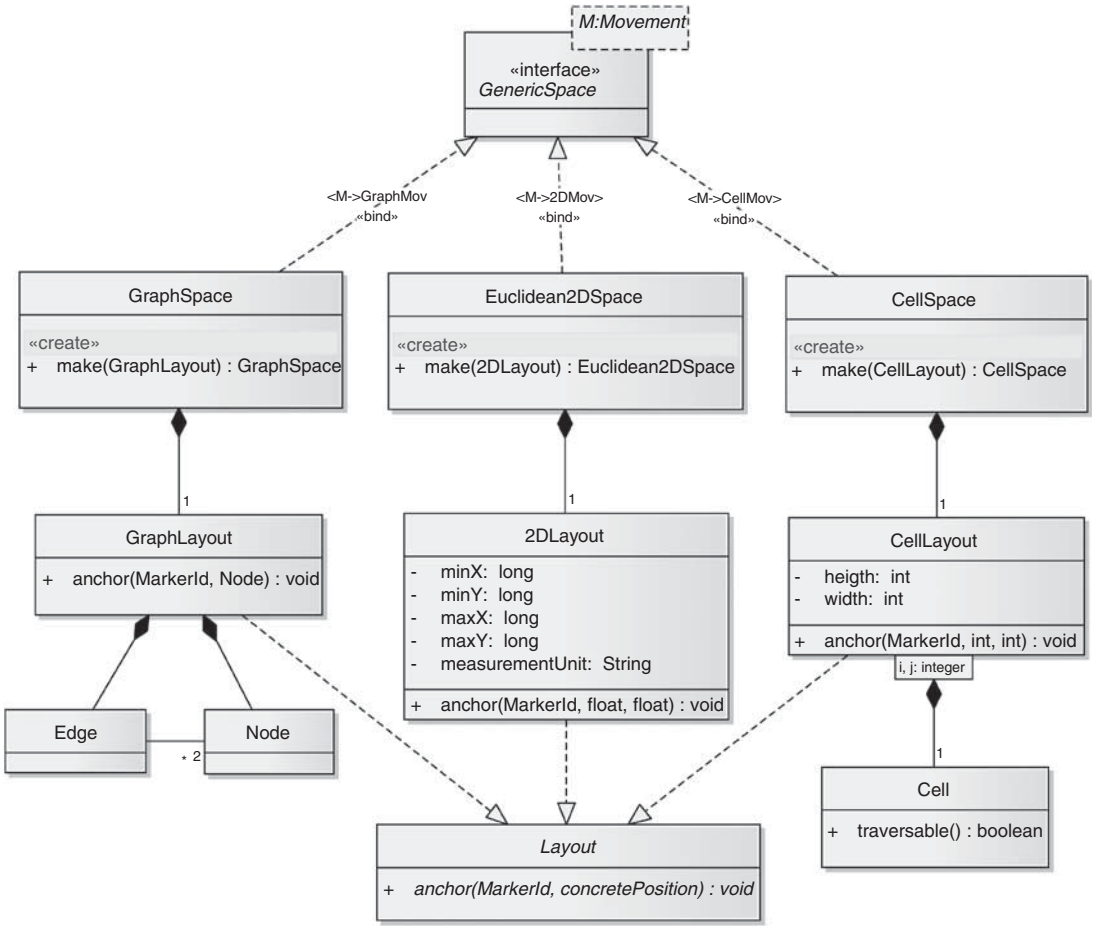

Figure 10 Space layout

typically exposes an 'ORIGIN' marker, which is anchored to a concrete location represented by a pair of float values. A graph space representing a building might expose an 'ENTRANCE' marker, which is anchored to the concrete implementation of a node.

5.4. Example

The snippets of pseudo-code in this section extend the example by highlighting marker usage and layout-related issues. The extended example introduces an implementation-aware agent (Figure 11) that plays the role of space factory by defining the layouts of the involved spaces and by instantiating them. It also defines the well-known markers by associating symbolic identifiers to graph nodes. Finally, it defines the space mappings.

Supervisor and coarse grain planner (Figures 12 and 13) are modified to exploit markers.

The supervisor (Figure 12) looks similar to the basic supervisor of Figure 7. However, there is a significant difference. In fact, the supervisor is no longer in charge of maintaining associations from thematic entities (e.g., `Room`) and locations. It only reasons in terms of symbolic room names and is not aware of any spatial issue.

The coarse grain planner too (Figure 13) looks similar to the basic coarse grain planner of Figure 8. The major difference is that it gets a task definition in terms of symbolic room names ('ROOM_A' and 'ROOM_B' passed by the supervisor). The coarse grain planner verifies if they can be interpreted as marker identifiers, exploits them to get the `here` and `there` locations, then it behaves like the basic coarse grain planner.

```
public class spaceFactory {
    GraphLayout gLayout;        //layout of the graph space
    GraphSpace gSpace;          //graph space

    CellLayout cLayout;         //layout of the cell space
    CellSpace cSpace;           //cell space

    2DLayout 2dLayout;          //layout of the Euclidean space
    Eucidean2DSpace 2dSpace;    //Euclidean space

    SpaceMapping gcMapping;     //a space mapping

    private void defSpaces() {
        //instantiate gLayout...
        //...add nodes and edges...
        Node node1 = new Node();
        Node node2 = new Node();
        Edge edge1 = new Edge(n1, n2);
        gLayout.add(node1);
        //...
        //define well-known markers
        gLayout.anchor("ENTRANCE", n1); // building entrance
        gLayout.anchor("ROOM_A", n1);
        gLayout.anchor("ROOM_B", n2);
        //...
        //create a graph space with the proper layout
        gSpace = GraphSpace.make(gLayout);

        //instantiate cLayout...
        //...initialize the cell layout (100 X 90 cells)...
        cLayout.heigth(100);
        cLayout.width(90);
        //define which cells are not traversable
        cLayout.setNotTraversable(7,25); //obstacle
        //...
        //create a cell space with the proper layout
        cSpace = CellSpace.make(cLayout);

        //instantiate gcMapping...
        //...set source and target...
        gcMapping.setSource(gSpace);
        gcMapping.setTarget(cSpace); =
        //graph node n1 maps to cell (70, 20)
        gcMapping.setMapping(n1,70,20);
        //...

        //instantiate 2dLayout...
        //...set boundary (10 x 9) and measurement unit (meters)...
        2dLayout.setBoundary(0.0,0.0,10.0,9.0);
        2dLayout.setMeasurementUnit("meter");
        //...
    }
}
```

Figure 11 Space factory

```
public class EnhancedSupervisor {

    //thematic domain information
    class Room {
        String name;                    //symbolic name of the room
        Goods[] goods;                  //goods stored in the room
    }

    EnhancedCoarseGrainPlanner cplanner;  //the exploited planner

    Room roomA, roomB;                  //thematic domain entities

    public void initialise(){
        //...
        roomA.setName("ROOM_A");
        //...
    }

    //define the vehicle task
    public void planTask(){
        //...
        //tell the planner to compute a path between markers
        cplanner.defineGraphPath("ROOM_A", "ROOM_B");
        //...
    }
}
```

Figure 12 Enhanced supervisor

```
public class EnhancedCoarseGrainPlanner {
    GraphSpace mySpace;             //the graph space exploited by planner
    MarkerId[] wellKnownMarkers;    //the well known marker ids
    //...other declarations as in basic coarse grain planner

    public void initialise(){
        //...
        //get the well known markers defined by the space;
        wellKnownMarkers = mySpace.allMarkers();

        //initialise the marker denoting the vehicle position
        mySpace.defMarker("VEHICLE",mySpace.locationOf("ENTRANCE"));
    }

    // the planner operation invoked by a Supervisor
    public void defineGraphPath(String fromRoom, String toRoom) {

        //check if fromRoom and toRoom appear in wellKnownMarkers...
        //...get locations from markers by interpreting fromRoom
        //and toRoom as markerIDs
        Location here = mySpace.locationOf((MarkerId)fromRoom);
        Location there = mySpace.locationOf((MarkerId)toRoom);
        Location vehicle = mySpace.locationOf("VEHICLE");

        //compute the path from vehicle to here;
        //add the path from here to there;
        //pass the cell path to the medium grain planner

        // update the vehicle marker
        mySpace.moveMarker("VEHICLE", there);
    }
}
```

Figure 13 Enhanced coarse grain planner

The coarse grain planner also defines and moves a 'VEHICLE' marker to persistently keep track of the current location of the vehicle, which is assumed to be initially at the location denoted by 'ENTRANCE'.

The description of the space factory is purposely verbose to highlight how different space models are characterized by defining specific layouts in different ways. This is unavoidably an implementation-dependent issue, but is encapsulated inside the space factory and does not affect the other space-aware agents.

Domain-aware agents (e.g., the supervisor) reason about pure thematic domain entities in terms of symbolic identifiers (e.g., 'ROOM_A') without dealing with space-related issues. For example, the supervisor decides that some goods should be moved from 'ROOM_A' to 'ROOM_B' without considering any spatial issue. In fact, the supervisor (Figure 12) does not exploit any space-related operation.

The coarse grain planner bridges thematic and spatial issues by exploiting a correspondence between symbolic names of thematic entities and marker identifiers. It gets 'ROOM_A' and 'ROOM_B' as call parameters from the supervisor. On the other hand, from its space it gets the set of ids of well-known markers through the `allMarkers()` operation. If 'ROOM_A' and 'ROOM_B' belong to that set, they can be properly interpreted as marker identifiers and exploited to get the corresponding locations `here` and `there`. Thereafter, the coarse grain planner reasons in terms of locations, as in Figure 8.

Note that the factory associates two markers, 'ROOM_A' and 'ENTRANCE', to the same node, N_1 (see Figure 2), with different meanings, namely 'this denotes a room' and 'this is the entrance of the building'. In a similar manner, the coarse grain planner exploits the 'VEHICLE' marker with the meaning 'this denotes the current position of the vehicle'. In general, several markers may be associated to the same location to model different domain concepts.

5.5 Space-aware architecture

Three major roles are involved in the definition, parameterization and use in the spatial concepts. The three roles correspond to activities that are well distinguished from the ontological, organizational and technological points of view:

- *Defining a space model* means to identify its conceptual properties, possibly in mathematical form, and to translate them into a proper implementation of its operations. This implies the full

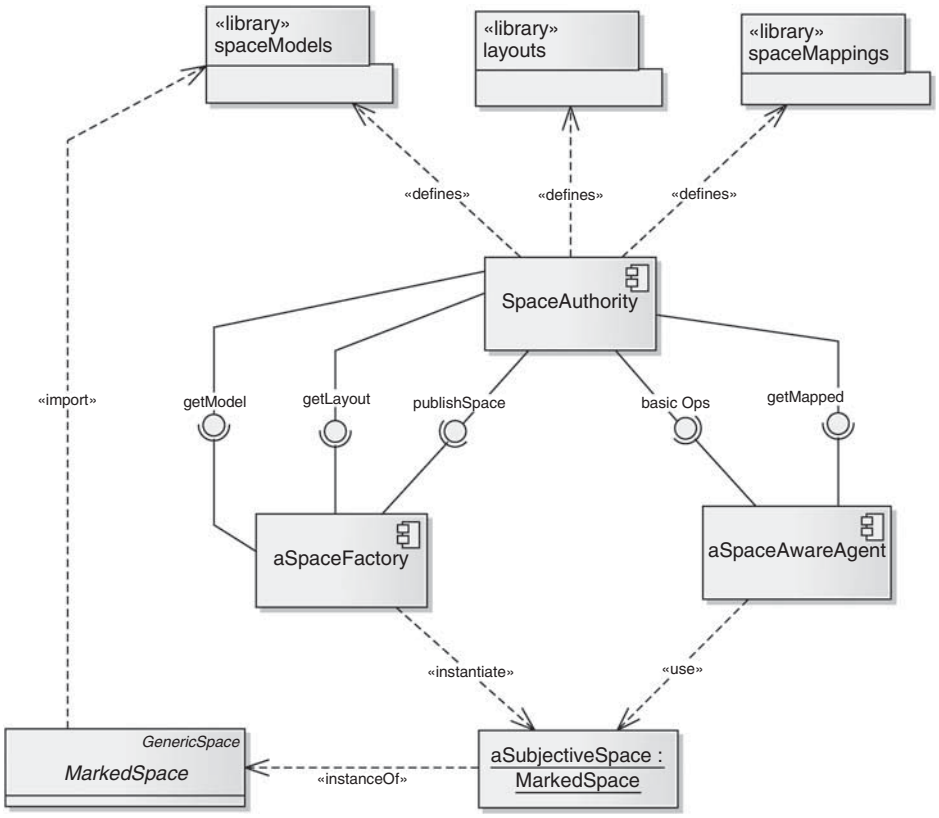

Figure 14 Space aware architecture

understanding of both conceptual properties and concrete implementation of the space model. In practice, this role (which, in the example, is played by the authors of this paper) has the responsibility of designing and implementing a subclass of `GenericSpace` and the proper subclasses of `Layout`.

- *Building a space* means to choose a space model and to instantiate it by specifying its layout. This role (played by space factory in the example) implies the understanding of a concrete layout representation for the selected model.
- *Using a space* means to reason about spatial issues by exploiting the basic spatial operations in terms of locations and movements. This role (played by planner and controller in the example) implies the need to know how to specify movements for the space at hand.

A single agent may play all the three roles above. Such an approach leads to the definition of a *subjective space*, which is defined and exploited by one individual agent. The problem becomes more complex as soon as several cooperating agents have to share a consistent visibility of one or more spaces. This requires an overall *space-aware architecture* supporting the capability of *sharing* space models and concrete spaces among cooperating agents.

Figure 14 gives the flavour of a space-aware architecture, in which specific components reify the above roles.

A well-known *space authority* is the reference for any space-aware agent. It is in charge of defining and maintaining libraries of space models, layouts and space mappings.

The space authority exposes interfaces that allow *space factories* to import space models and layouts (i.e., class definitions). A space factory exploits models and layouts to instantiate subjective spaces. Spaces are subjective because they are known only by the instantiating factory, which builds them according to subjective criteria. However, a factory may *publish* a space it defined through an interface provided by the space authority.

A *space-aware agent* exploits published spaces through the basic spatial operations. In particular, Figure 14 highlights that an agent may ask the authority for mapping between locations defined in different spaces.

This conceptual architecture can be deployed in different ways. The most obvious solution is to turn the space authority into a service, for example, a Web service. However, other solutions can be devised: for example, basic space operations can be realized by methods of classes imported by space-aware agents, which can operate locally on subjective spaces. The authority is involved only if space mappings are required. Such issues are the subject of ongoing research and experimentation activities.

6 Conclusions and future directions

This paper discussed, through a simple example, how basic software engineering principles can be exploited to identify a coherent set of spatial abstractions and to devise a space-aware architecture. The results can be summarized as follows:

- Thematic and space-related aspects are separated by carefully distinguishing between symbolic identifiers, opaque locations and concrete space layouts.
- Space-aware agents rely on an operational view of the space, which is defined in terms of opaque locations and of movements between them.
- The concrete layout of a space is specified through a set of operations, the use of which can be restricted to privileged implementation-aware agents.
- The definition of space models is up to a space authority, which is fully aware of both conceptual and concrete space-related issues.

From the software engineering point of view, the result is a set of design and architectural patterns that may help both the development of space-aware systems and the integration of heterogeneous space-aware applications.

The proposed approach is being exploited in two research projects. In the InSyEme project (Integrated System for Emergency), funded by the Italian Research Ministry, spatial abstractions are treated as an orthogonal aspect encompassing all the layers of a critical distributed system. This supports an effective management of the communication infrastructure by integrating different space models related to network topology, mobile users localization and user roles in an organizational space.

The BIGI project (Bicocca Integrator of Georeferenced Information), carried on at the University of Milano—Bicocca, focuses on the integration of heterogeneous georeferenced databases relying on different spatial models. The aim is to support the cross-domain statistical analysis of data related, in particular, to sociological and environmental issues.

From an ontological point of view, some issues deserve a deeper analysis: first, the *operational* view of the space; second, the idea that spaces are intrinsically *subjective*, as their definition is up to the agents; third, as a consequence, the conjecture that space ontologies are meaningful as long as they are related to roles and interaction patterns in a *cooperative context*. Agents playing different roles cooperate to get an agreement about space models and to get visibility over shared spaces.

The authors hope that these issues will contribute to a fruitful osmosis between ontology- and engineering-oriented research areas.

Acknowledgement

This work is part of the InSyEme (Integrated System for Emergency) project. InSyEme is funded by MIUR (Italian Ministry of Education, University, and Research) under the FIRB Programme for supporting basic research, Grant no. RBIP063BPH.

References

Bass, L., Clements, P. & Kazman, R. 2003. *Software Architecture in Practice*, 2nd edn. Addison-Wesley.

Bateman, J. & Farrar, S. 2004a. Modelling models of robot navigation using formal spatial ontology. In *Spatial Cognition IV: Reasoning, Action, Interaction*, Frauenchiemsee, 11–13 October 2004, Freksa, C., Knauff, M., Krieg-Brückner, B., Nebel, B. & Barkowsky, T. (eds). Springer, 366–389.

Bateman, J. & Farrar, S. 2004b. *Spatial Ontology Baseline*. Collaborative Research Center for Spatial Cognition, University of Bremen (I1-[OntoSpace]:D2, SFB/TR8).

Bateman, J. & Farrar, S. 2004c. Towards a generic foundation for spatial ontology. In *Formal Ontology in Information Systems*, 4–6 November 2004, Varzi, A. C. & Vieu, L. (eds). Torino, 237–248.

Borgo, S., Guarino, N. & Masolo, C. 1996. Stratified ontologies: the case of physical objects. In *Proceedings of the Workshop on Ontological Engineering at ECAI*, Vet (ed). Budapest, 5–16.

Clements, P. 2008. *Origins of Software Architecture Study* [online]. Available from `http://www.sei.cmu.edu/architecture/roots.html` (accessed 18 June 2008).

Egenhofer, M. & Rodrguez, A. 1999. Relation algebras over containers and surfaces: An ontological study of a room space. *Spatial Cognition and Computation* **1**(2), 155–180.

Fonseca, F. T., Egenhofer, M. J., Agouris, P. & Camara, G. 2002. Using ontologies for integrated geographic information systems. *Transactions in GIS* **6**(3), 231–257.

Frank, A. U. 2001. Tiers of ontology and consistency constraints in geographical information systems. *International Journal of Geographical Information Science* **15**(7), 667–678.

Ghezzi, C., Jazayeri, M. & Mandrioli, D. 2003. *Fundamentals of Software Engineering*, 2nd edn. Prentice-Hall.

Gibson, J. 1977. The theory of affordances. In *Perceiving, Acting, and Knowing: Toward and Ecological Psychology*, Shaw, R. & Brandsford, J. (eds). Erlbaum, 62–82.

Howarth, R. J. 2005. Spatial models for wide-area visual surveillance: computational approaches and spatial building-blocks. *Artificial Intelligence Review* **23**(2), 97–155.

Kuipers, B. 2000. The spatial semantic hierarchy. *Artificial Intelligence* **119**(1–2), 191–233.

Marchese, F. M. 2005. A reactive planner for mobile robots with generic shapes and kinematics on variable terrains. In *Proceedings of the 12th International Conference on Advanced Robotics*. IEEE Robotics and Automation Society, 23–30.

OMG. 2008. *UML 2.1.2 Superstructure and Infrastructure* [online]. Available from `http://www.omg.org/technology/documents/formal/uml.htm` (accessed 18 June 2008).

Parent, C., Spaccapietra, S. & Zimányi E. 1999. Spatio-temporal conceptual models: data structures+space+time. In *Proceedings of the 7th ACM International Symposium on Advances in Geographic Information Systems*. Kansas City, 26–33.

Parnas, D. L. 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* **15**(12), 1053–1058.

Pelekis, N., Theodoulidis, B., Kopanakis, I. & Theodoridis, Y. 2004. Literature review of spatio-temporal database models. *Knowledge Engineering Review* **19**(3), 235–274.

Perry, M., Hakimpour, F. & Sheth, A. 2006. Analyzing theme, space, and time: an ontology-based approach. In *Proceedings of the 14th Annual ACM International Symposium on Advances in Geographic Information Systems*. Arlington, 147–154.

Smith, B. & Grenon, P. 2004. The cornucopia of formal–ontological relations. *Dialectica* **58**(3), 279–296.

Software Engineering Institute, Carnegie Mellon. 2008. *Published Software Architecture Definitions* [online]. Available from `http://www.sei.cmu.edu/architecture/published_definitions.html` (accessed 18 June 2008).

Tarr, P., Harrison, W., Ossher, H., Finkelstein, A., Nuseibeh, B. & Perry, D. 2000. Workshop on multi-dimensional separation of concerns in software engineering (workshop session). In *Proceedings of the 22nd International Conference on Software Engineering*. Limerick, 809–810.

Tisato, F., Micucci, D., Adorni, M. & Cirasa, E. 2007. Architectural abstractions for space awareness. In *Proceedings of the 2nd International Workshop on Ontology, Conceptualization and Epistemology for Software and Systems Engineering*, Sicilia, M., Micucci, D. & Sartori, F. (eds). Centro Copie Bicocca. Available from `http://www.lintar.disco.unimib.it/ONTOSE07/ONTOSE07-Proceedings/pdf/Micucci.pdf` (accessed 18 June 2008).

Yuan, M. 1996. Modeling semantical, temporal and spatial information in geographic information systems. In *Geographic Information Research: Bridging the Atlantic* Craglia, M. & Couclelis, H. (eds) Taylor & Francis, 334–347.