1

Object Oriented Approach and UML

The Object Oriented (OO) approach is not really new; it has been known and used since the 1990s [MEY 88, MEY 97]. The OO approach uses Unified Modeling Language (UML)[1] diagrams [RUM 98] for documenting the analysis and design stage, and languages supporting the OO paradigms (*C++*, Java, C#, etc.) for the programming stage. Some UML tools (commercial or open source) can generate more or less complete *C++* code from design specifications.

In the following, details on UML graphical notations will be presented as necessary in order to illustrate OO items. Section 1.3 presents some higher level considerations on UML diagrams used in OO analysis and design.

1.1. Object Oriented (OO) paradigms

We present hereafter some of the main paradigms encountered within OO analysis and design, class, object, inheritance, encapsulation, polymorphism and genericity:

Class: a class is a *model* that can represent any entity or concept identified at the analysis or design stage. The class models the *data* structure and provides the *behavior* of the associated concept or entity. Class data are called

1 www.uml.org.

attributes. The class behavior is defined by member functions called *methods*. Class methods are allowed us to play with the class data.

Within the DEM context, we need classes such as *DiscreteElement*, *Spring*, *Contact*, *Acceleration*, *Position*, *Vector*, *Quaternion*, *Point*, *DiscreteDomain*, etc.

Programming languages supporting the `class` keyword consider a class as a new type (a *user type*).

Object: an object is the result of a transformation operated on a class called *class instantiation*. The instantiation is the mechanism through which we give concrete values to the class attributes in order to construct an object. Each object has the behavior of its class, but owns its specific individual set of attribute values.

Most OO programming languages provide a special class method called the *constructor* used to instantiate objects from classes. Another special class method called the *destructor* can be used to make an object die (technically speaking, it is removed from the computer memory). The type of an object is the class it has been instantiated from.

Inheritance: inheritance is *the* mechanism that gives OO models their power, scalability, extendibility and reusability. Due to inheritance, we can design base classes factorizing data and behaviors that can be used by derived classes. Derived classes can simply use behavior of their base class(es) through the base class methods, or redefine a specific behavior by redefining some of the base class methods at their level.

All languages that are said to be OO should give syntactic elements to implement inheritance. The *C++* language proposes a rich support of inheritance (multiple inheritance, public/protected/private inheritance, abstract class, virtual methods, purely virtual methods, etc.). Inheritance also avoids the copy–paste programming drawbacks.

Encapsulation: class data can be protected from hazardous manipulations by qualifying them as *private* or *protected*: a class attribute that is private

(respectively, protected) cannot be manipulated from outside the class (respectively, from outside a derived class).

Hiding class attributes as private or protected data allows us to design and write more robust software: access to class attributes is achieved by using some special class methods that control what is done with the class data. This mechanism avoids a lot of software bugs occurring when direct access to data is allowed, potentially leading to data corruption.

Polymorphism: OO polymorphism has two main forms: dynamic or static. At the design stage, the (dynamic) polymorphism is mainly a consequence of inheritance: a common behavior defined in a base class can be redefined specifically in each derived class. At (software) run time, the type of the derived class will dynamically decide which implementation of the behavior will be launched.

At the coding stage, languages such as *C++* also offer a (static) polymorphism relying on the overloading mechanism: the same identifier (for instance, language operator name, function name or class method name) can behave in different ways depending on the type and number of the given arguments.

Genericity: genericity is the ability to make some design items depend on a *template parameter* that generally represents a class or a type. It is an important key in designing OO models with a high level of abstraction.

From the programming point of view, genericity can take various formulations, depending on the language used. For the *C++* language, generic classes are those which depend on a template parameter. A typical example is the class container of "something", where "something" is the template parameter that can represent any class. You can have a container of "int", a container of "Points", a container of "DiscreteElements" etc. Generic behavior for "something" is coded as a generic *C++* source code. Each time the generic class is used with a given instantiation of the template parameter ("int", "Point", "DiscreteElement", etc.), the *C++* compiler automatically generates and compiles a specific version of the source code obtained by replacing all the occurrences of the text "something" in the generic code by the text "int", "Point", "DiscreteElement" or anything else.

Genericity is a very valuable quality of programming languages, when possible. It saves a lot of developers time because the developer only writes the generic code in a single version. It is also a key factor for software scalability.

1.2. OO analysis and design

The main goal of the OO approach (analysis or design) is to capture concepts and entities of the studied system in order to associate classes to each concept or entity of the studied system. Once the classes have been found, the OO model must be completed to find out the relations between classes.

So, a major task when designing an OO solution is to establish the relations between classes that have been found at the first step to model correctly the studied system. Of particular interest among the various type of relations that can be used are the association (with its possible association class), inheritance, aggregation and composition.

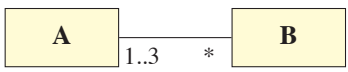
1.2.1. *Association*

The first relation we present here is the simple *association*: it is a very commonly used relation and despite its simplicity, it is the key to a fundamental mechanism: when class A is associated with class B, A knows that B exists and A can make B to do something (in OO context, we say that A can *send a message* to B).

Sending messages between classes is a crucial mechanism to make the OO model do something.

Association is the simplest relation: class A knows class B and reciprocally. A and B can exchanges messages...

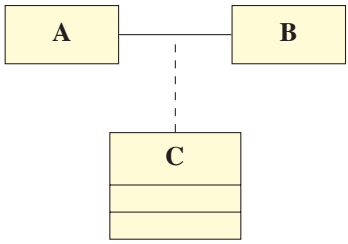
Unidirectional association: class A knows class B, but B does not know A.



Mutiplicities: class A is associated to 0 or more occurences of class B. Class B knows 1,2 or 3 occurences of class A.

1.2.2. *Association class*

When an association between two classes carries some data and has a behavior, a class can be used to model the data and behavior of the association : it is called an *association class*.



The association class C models data and behaviour of the association between classes A and B, if any.

1.2.3. *Inheritance*

As we have already seen, inheritance is a major paradigm of the OO approach. A very common usage of inheritance within OO analysis and design is when you come to say that "A *is a* B" like in the phrase "a cube *is a* volume shape". It means that all what a volume shape can do, a cube can do the same (or can do it differently, or better, if the behavior is redefined for the cube). In this situation, you would say that the `Cube` class uses the `VolumeShape` class.

The graphical UML representation is a white arrow:

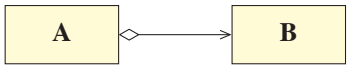
Class B inherits from classs A: A is the base class, mother class...
B is the derived class.
B *is a* A.

1.2.4. *Aggregation*

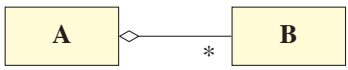
The aggregation is a relation in which one class *contains* other classes. It is often used in OO analysis and design situations where an entity *is made of* or *contains* one or many other entities.

The UML graphical representation of the aggregation is a line between the container and the components, with a white diamond on the container side. Multiplicities can be used to indicate the number of occurrences of each class.

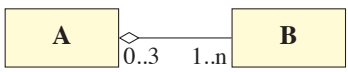
Class A contains class B.



Unilateral aggregation: class A contains class B, but class B has no relation with class A.



Class A contains zero or more occurences of class B.



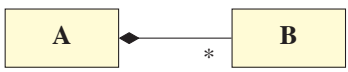
Class A contains one or more occurences of class B. Class B is contained by 0, 1, 2 or 3 occurences of class A.

1.2.5. *Composition*

Composition is often introduced as a strong form of aggregation, with an impact on the lifetime of the components: when the object holds the composition dies, all the components are also deleted. Unlike aggregation, components of a composition make part of one and only one composition. The UML graphical representation of the composition is a line between the container and the components, with a black diamond on container side. Multiplicities can be used to indicate the number of occurrences of each class.

Class A is composed of class B.

Unilateral composition: class A is composed of class B, but class B does not know class A.

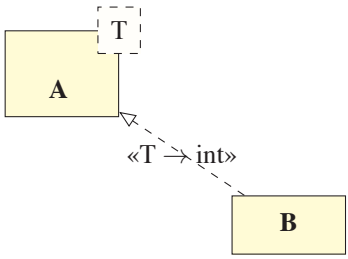


Class A is made of zero or more occurences of class B.

1.2.6. *Genericity with the template classes*

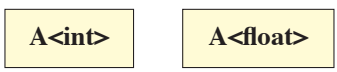
Genericity is a very powerful paradigm of the OO approach. A generic class is parametrized with a template parameter that can represent a type, a class or an integral type. A generic class cannot be instantiated as this: we must give a "value" to the template parameter (in other words, we must *instantiate* the template parameter) to obtain a concrete class that can be instantiated in objects.

UML provides a graphical representation of a template class where the template parameter(s) is (are) drawn in a little box at the top right of the class box.



The generic class A, with the template parameters `T`. A generic class cannot be instantiated.
The concrete class B instantiated the template parameters `T` to the type `int`. B can be instantiated, it corresponds to the type `A<int>`.

The *C++* compiler generates *on the fly* the source code of the `A<int>` class by substituting each occurrence of template parameters `T` by `int` in the generic source code file of the `A` class. If the implementation of a method of the template `A` class cannot simply be obtained by replacing `T` by `int`, the developer has to provide himself a *specialized* version of these methods for the `int` case.



`A<int>` is a type (the name of a class), `A<float>` is a different type. They both come from the instanciation of the template parameter `T` of the `A` class, but they are two different concrete classes.

Generic classes are often used to model *containers of something*. Some famous examples in *C++* are the containers of the standard template library (STL) such as, for example the `vector`, `list` and `map` classes. Within the *GranOO* workbench, The `libDEM` library provides the class `SetOf` that models the concept of "set of something" (`SetOf` class uses the `vector` STL class). `SetOf` objects are useful to store discrete elements, bonds, springs, etc. We

can retrieve a `SetOf` by its name, we can scan its elements in a loop, we can add/remove items, etc. (more on `SetOf` is given in section 3.2.1).

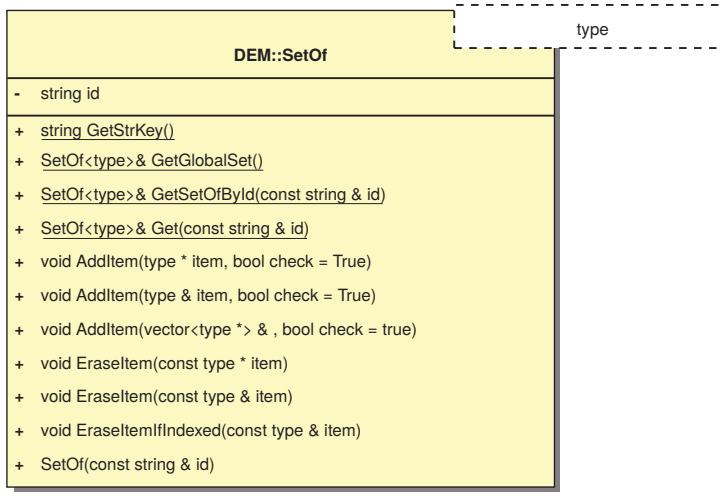


Figure 1.1. *The template class `SetOf` (from the `libDEM` library)*

1.2.7. *Encapsulation and class interface*

As already seen, software robustness can be improved due to the encapsulation mechanism: the "iceberg metaphor" is often used to present objects like icebergs floating at the surface of the water:

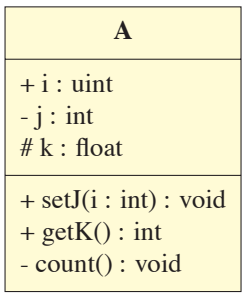
– all what is visible above the water (attributes and methods) make the *object interface*, its **public** part;

– all what is under the surface (attributes and methods) make the *object implementation*, its **private** part.

When using an object, as a user, a program, or another object, etc., we are only concerned with the interface of the object, and not with the internal details explaining how the object treats our request. When an object evolves with time (under the maintenance of the source code or the evolution of the OO design), its interface must remain stable to ensure the object perennity. This is a real deal in OO design and development: design robustness relies on the objects

interface, so if we want to develop a robust software we must play with the encapsulation. This advice gives a way of coding also known as *data hiding*.

UML provides standard artifacts to show the visibility of the members (attributes or methods) of a class:



+ before a class member means a **public** member, belonging to the interface of the class.

- before a class member means a **private** member, belonging to the private part of the class

before a class member means a **protected** member, only accessible by derived classes, if any.

1.3. UML diagrams

UML diagrams are the documentation of OO analysis or design. UML is a standard that defines many different diagrams: UML2 defines 14 diagrams (www.uml.org) that can be classified into static (structural) and dynamic diagrams:

The static diagrams are related to the data structure of the studied system:

– *Class* and *Object diagrams* describe the different entities revealed by the OO analysis or design, their structure, data, behavior and inter-relationships.

– *Component, Deployment, Package* and *Composite diagrams* are useful for managing the technical aspects of software development.

The dynamic diagrams, related to the behavior of the studied system, are:

– *Use case, Activity* and *State machine diagrams*;

– *Sequence, Communication, Interaction and Timing diagrams* are useful for describing time varying aspects of the studied system.

The UML diagrams are presented through *views*, which can show more or less details on the entities they cover. For instance, at global design, the class diagram views can only show the name of the classes and their relations.

Details on the attributes or methods of the classes are not shown, because we just want to represent the main entities and their relations.

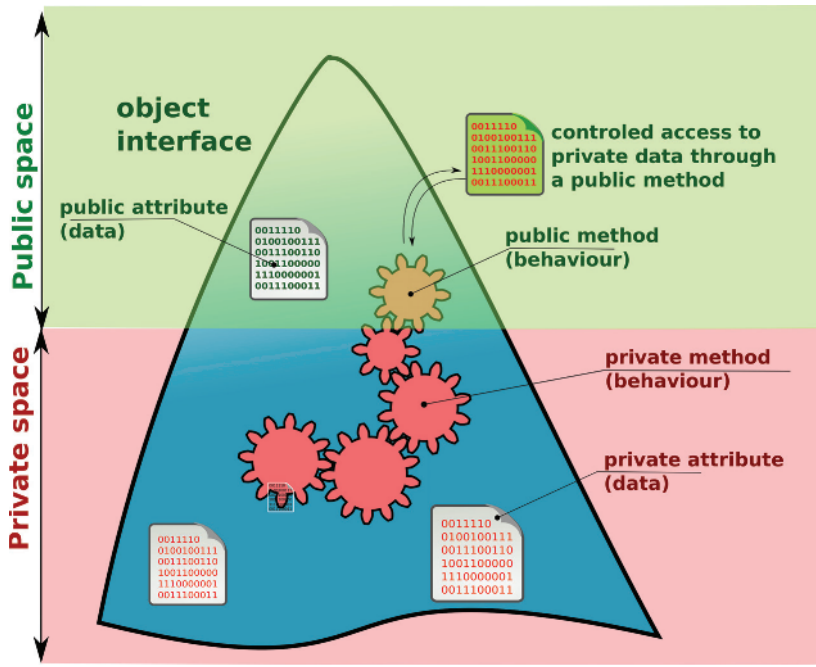


Figure 1.2. *Encapsulation mechanism*

For the detailed design point of view, on contrary, we are concerned with fine details on every class, and it can be useful to edit the detailed view with only one class per view, showing extensive information on each class: attribute names, quality and types as far as methods arguments and return type.

We present in the following the two main used diagrams for *GranOO* documentation: class diagram and sequence diagram.

1.3.1. *Class diagram*

A class diagram represents the classes with their attributes and methods, and the relations between classes. It is a static diagram which is very useful to represent globally the relations between classes, and specifically the attributes and methods of a given class.

Figure 1.3 shows a global view of the template class `DiscreteElementShaped` and the hierarchy derived from the `Interaction` class. The template parameter `shape` is used to design `DiscreteElementShaped` as a class that depends on another: the one given by the template parameter. In order to use `DiscreteElementShaped`, we must instantiate a type for the template parameter: this gives, for example, the class `DiscreteElementShaped<Geom::Sphere>`, where the template parameter `shape` is instantiated to the type (i.e. the class) `Geom::Sphere`. The template parameter `N` represents the space dimension. Currently, *GranOO* implements the instantiation of this template parameter with the value `_3D`, a constant integer given in the libGeometrical library.

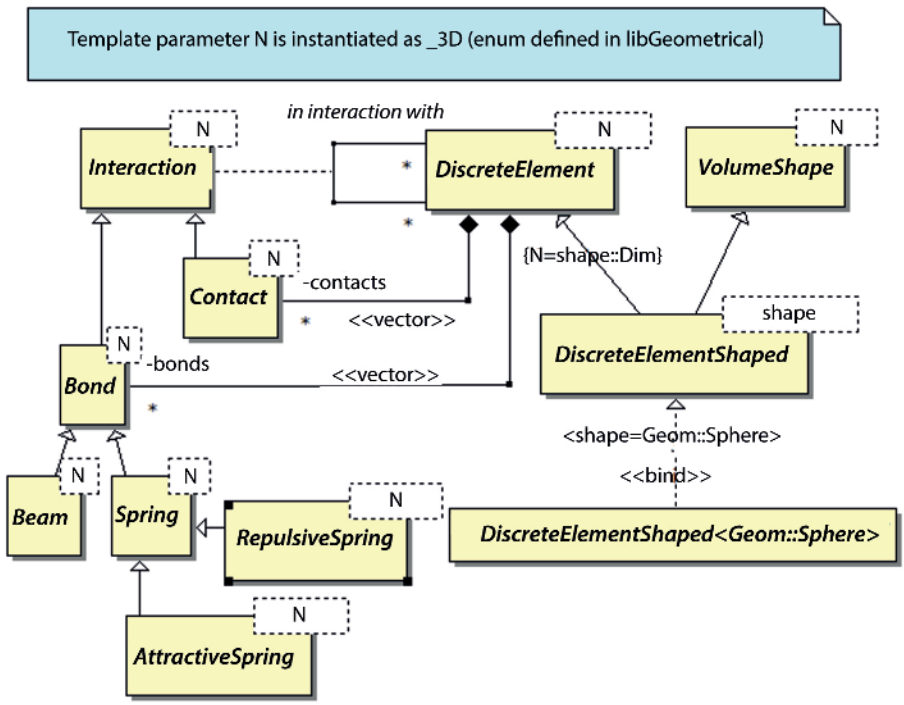


Figure 1.3. *Example of global class diagram view showing the class `DiscreteElementShaped` (from the libDEM library) and the `Interaction` hierarchy*

Figure 1.4 shows a global view of the template class `DiscreteElement`, from the libDEM library. Different levels of details may be shown on a class diagram view, depending on its usage. At detailed design stage, it may be useful to see the list of the attributes and methods of a class, but without details on the methods arguments and return types (see Figure 1.5(a)). If the detailed view is to be used by developers as a specification of the methods signature, a more detailed view can be given (see Figure 1.5(b)). Class diagram view of the class DiscreteDomain where only the list of attributes and methods is shown.

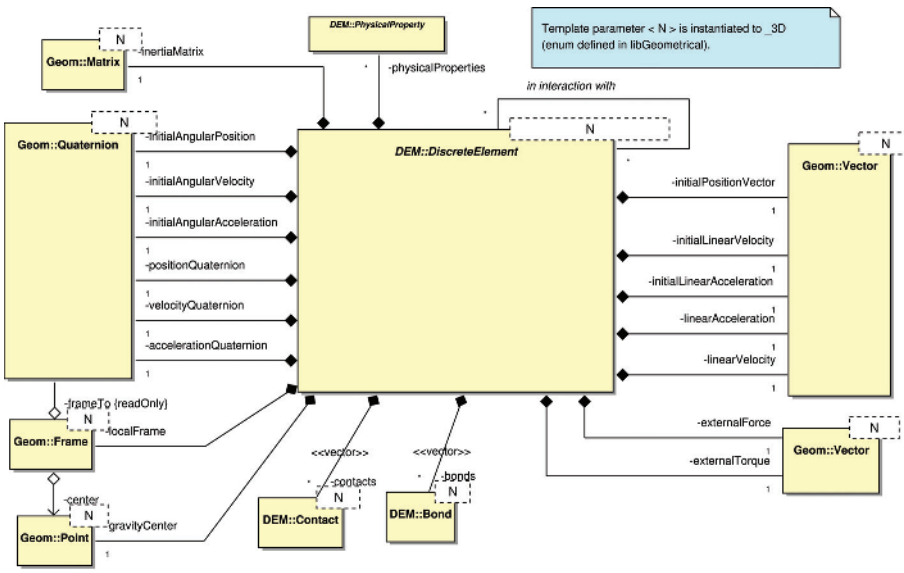
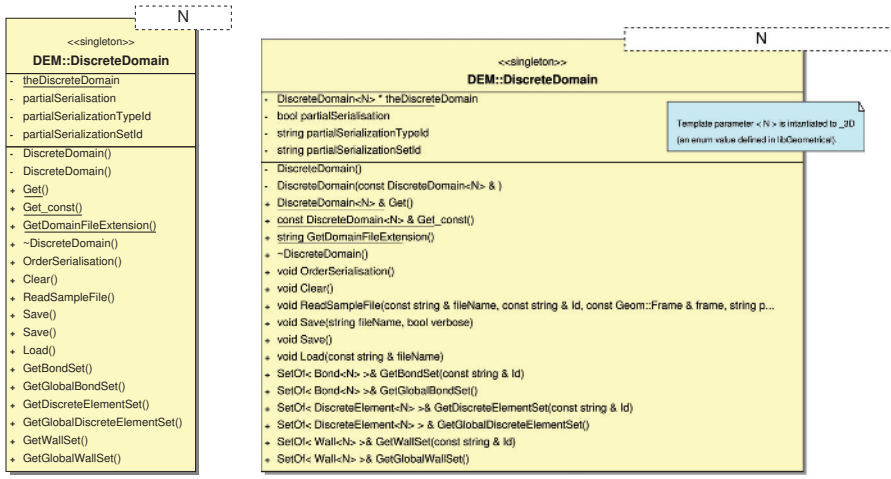


Figure 1.4. *Example of class diagram view showing the class* `DiscreteDomain` *(from the libDEM library) and its relations with these classes Point, Vector, etc. (from the libGeometrical library)*

1.3.2. *Sequence diagram*

A sequence diagram represents a dynamic view of the system where the time flows from top to bottom. The objects are placed in the diagram and can exchange messages by using the methods of the classes they come from. A set of objects exchanging messages implements a function of the system.



a) Simple list b) Full detailed methods signature

Figure 1.3. *Class diagram view showing the list of methods of the class DiscreteDomain (from the libDEM library)*

UML graphical decorators can be used to implement alternatives (the equivalent of an if/else), loops, options, etc. as illustrated in Figure 1.6. This sequence diagram corresponds to the overall operations executed by a Discrete Element Method (DEM) simulation build with the *GranOO* workbench. The leftmost vertical line corresponds to the `ComputeProblem` singleton class. Its static method `Run` is called by a `main` function. Any executable application build within the *GranOO* workbench implements a `main` function as follows:

```
#include "CommonLibsNameSpace.hpp"
#include "libDEM/SRC/ComputeProblem.hpp"

int main(int argc, char * argv[])
{
  DEM::ComputeProblem<Geom::_3D>::Run(argc, argv);
  return 0;
}
```

The static `Run` method is called by the `main` function using the *C++* syntax `ClassName::staticMethodName(...)`. Then, it calls another static method

of the `ComputeProblem` class: the `Get` method. `Get` creates the object `theComputeProblem` as the sole instance of the `ComputeProblem` class: it is illustrated in Figure 1.6 as the rightmost vertical line. A simple well-known mechanism (the *singleton* pattern design) ensures the unicity of the instantiation of the `ComputeProblem` class.

The sequence diagram shows how the `Run` method calls successively the right methods of the object `theComputeProblem` to run the preprocessing plugins, the processing plugins of the main loop and finally the postprocessing plugins.

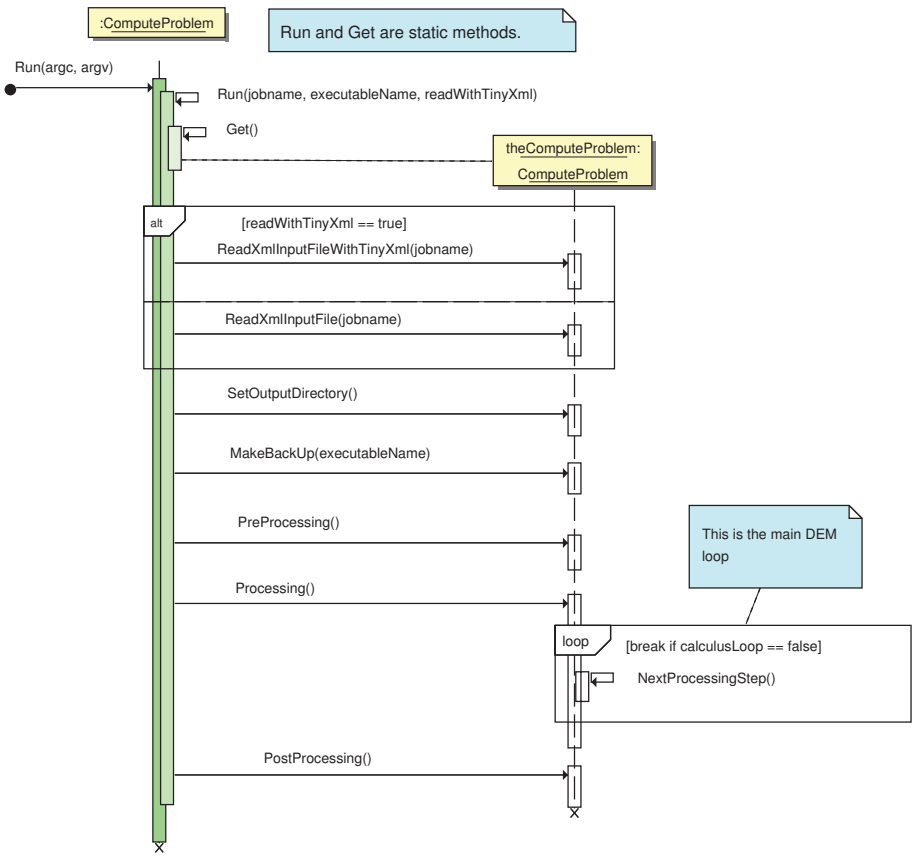


Figure 1.6. *Sequence diagram involving the `ComputeProblem` singleton class (from the libDEM library) and the object `theComputeProblem` instantiated from this class*