Comput Mech (2005) 36: 85–99
DOI 10.1007/s00466-004-0612-y

ORIGINAL PAPER

C.-T. Yang · S.-H. Hsieh

# An object-oriented framework for versatile discrete objects simulation using design patterns

Received: 30 May 2004 / Accepted: 5 July 2004 / Published online: 14 April 2005


**Abstract** This paper proposes a framework for versatile discrete objects simulation. The framework, named VEDO, is developed using object-oriented technology with design patterns. VEDO is capable of handling simultaneously discrete objects of various shapes and various mechanisms of interactions between discrete objects. It also has great flexibility in facilitating additions of new discrete object shapes and solution algorithms for discrete object interactions. Based on the proposed framework, a discrete objects simulation system, named Knight&Anne, has been implemented in C++ in this study. In addition, some application examples are given to demonstrate the capability and flexibility of the framework.

**Keywords** Discrete objects simulation · Object-oriented design · Design patterns · Particle simulation

## 1 Introduction

Discrete objects simulation is widely used in science and engineering, such as, for example, in studies of the mechanics of rocks and soil [10], simulations of the flow of concrete [2][14] and colloidal dynamics [7][13]. A simulation system, which models discrete objects and applies physical equations of motion to simulate their dynamic behaviors, is called a Discrete Object Simulation System (DOSS) hereinafter.

The methodology employed by DOSS usually has two major parts. One is the construction of models of discrete objects with various geometries. The other is the modeling of the mechanisms of interactions among discrete objects, to mimic the behavior of real materials. In some applications, because the simulation results are highly sensitive to the shapes of discrete objects, DOSS should be able to handle the complexity of the geometries of discrete objects. Therefore, research in this field progressed from 2D geometrical models to 3D models and from simple spherical models to complicated polyhedral models. In the other applications, DOSS should be able to handle various mechanisms that operate among various discrete objects when materials are composite. Consider, for example, the simulation of concrete flowing [2][14]. Concrete comprises of two materials - discontinuous aggregates and continuous mortar. Clearly, the mechanism of interactions between the aggregates is different from that between the aggregates and the mortar.

However, most existing DOSS packages usually can only handle either various shapes or various mechanisms but not both. For example, PFC$^{3D}$ [6], a popular commercial package, can handle various mechanisms but has only spherical discrete objects in 3D. Most of DOSS packages developed by academic researchers, e.g., DEM3D [3] and the one by Hong [7], usually meet only the requirements for carrying out their research work and often can not be easily extended (if not impossible) to handle more object shapes and interaction mechanisms for more general applications.

To build a versatile DOSS, which is one that can simultaneously handle various shapes and various mechanisms, a flexible underlying software framework is very important. The framework must be flexible enough to incorporate systematically and correctly numerous mathematical models for representing discrete objects and solution algorithms for interactions among discrete objects. Therefore, this study employs

The support from the National Science Council of Republic of China under Grant No. NSC91-2211-E-002-097 and NSC91-2211-E-399-001 is greatly appreciated. In addition, the authors would like to thank Prof. Chuin-Shan Chen of National Taiwan University for his review and suggestions on the design and implementation of the proposed framework as well as Mr. Li-Shin Lin and Mr. Jenn-Feng Li for their help on example studies.

C.-T. Yang · S.-H. Hsieh (✉)
Department of Civil Engineering,
National Taiwan University, Taipei 10617, Taiwan
e-mail: shhsieh@ntu.edu.tw

the object-oriented technology to develop such a flexible framework. The framework proposed is named VErsatile Discrete Objects Framework (VEDO). It is never the intention of this study to develop a general-purpose DOSS package. The objective of this study is to develop a flexible application framework so that engineers and researchers can easily build the application-specific DOSS they need. The focus of this paper is on the design of VEDO although some discussions on the implementation are provided.

VEDO consists of many designs presented in Design Patterns [5]. Design Patterns have been widely used for developing scientific software in recent years [21]. They are highly abstract designs that can help to clearly state the responsibility, the structure, the behavior, and the creation of individual class or whole software system in brief statements. Design patterns are also reusable designs that describe simple and elegant solutions to specific problems that appear repeatedly in object-oriented software design. To the best of the authors' knowledge, the use of Design Patterns in developing object-oriented discrete objects simulation framework has not been seen in the literature. Moreover, the major difference between the present object-oriented design and those presented previously, e.g., [3][19][20], is that the previous design emphasizes mainly on the polymorphism associated with the modeling of discrete objects with few discussions on the handling of interactions among discrete objects with many different geometric shapes. The present design emphasizes more on the polymorphism associated with interactions among discrete objects because it is much more complicated in nature when discrete objects of various shapes and various mechanisms of interactions between discrete objects are considered simultaneously in the simulation. In addition, object composition [5] instead of inheritance is employed in the present design to achieve interaction polymorphism and to reduce the implementation complexity.

This study first analyzes the requirements of VEDO and then constructs VEDO from currently existing design patterns. Unified modeling language (UML) [1] is used to describe the model of VEDO. As an effective means of specifying, visualizing, constructing and documenting the artifacts of a software system, UML is a collection of meaningful notations that model the structure and behavior of all classes and objects of a software system. Moreover, to verify and illustrate the functionality and flexibility of VEDO, this study uses VEDO to prototype a DOSS in C++, named Knight& Anne, which is then used for simulations of some application problems.

The rest of this paper is organized as follows. Section 2 analyzes the requirements of VEDO. Section 3 then presents the design of kernel classes. Next, in Sect. 4, the design of auxiliary classes is presented. Section 5 discusses briefly the implementation of Knight&Anne and provides some illustrative examples. Conclusions are finally drawn in Sect. 6.

## 2 Analysis of requirements

A DOSS is frequently employed to simulate the dynamic behavior of discrete objects by constructing geometric models and physical mechanisms. For example, in a simulation of granular aggregates in a sieve analysis, polyhedral solids are used to model the aggregates and the sieves; physical laws (e.g., the conservation of momentum) are also applied, as revealed in Fig. 1. The simulation of a sieve analysis must address four issues to approach the physical experiment:

(1) *Object Modeling*: how should geometric models of all discrete objects (mobile aggregates and immobile sieves) be built?
(2) *Contact Detection*: how should contact between any pair of discrete objects (e.g., aggregates vs. aggregates or aggregate vs. sieve) be identified?
(3) *Impact Solution*: how should the total external impact (force) applied to each discrete object over a particular time interval be obtained?
(4) *Object Motions*: how should the change of each discrete object's status, including position, velocity, orientation, and angular velocity, be determined according to the total external impact applied to it over a particular time interval?

In fact, the above four issues must be addressed regardless of what simulation a DOSS is applied. As well as considering the above issues, a versatile DOSS must manage multifarious algorithms. For example, an algorithm for detecting contact between spherical and cylindrical objects differs from that for detecting contact between spherical and planar objects. A versatile DOSS must determine what is the correct time and conditions

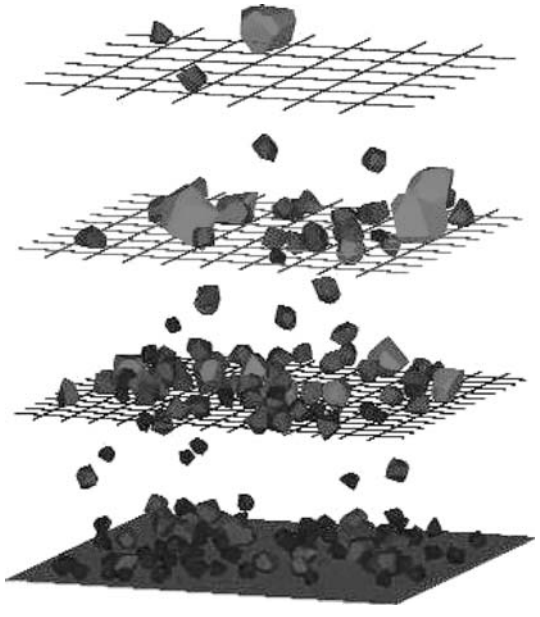


**Fig. 1** Simulation of a sieve analysis

for the execution of each algorithm if these three types of discrete objects are simultaneously present in a system. Since it is too difficult to consider all known shapes and mechanisms at the beginning stage of developing a versatile DOSS, the underlying framework of a DOSS should have the flexibility to integrate new functionalities at low cost. Moreover, although a DOSS usually repeats a certain sequence of tasks during each time interval (more discussions are provided in Sect. 4.3.2), some special applications may require special simulation procedure, e.g., Brownian dynamics [13]. Therefore, the underlying framework should also have the flexibility to incorporate new simulation procedures.

In summary, VEDO, the new framework to be developed in this study, should address the aforementioned four issues and meets the following requirements:

(1) Capable of handling simultaneously discrete objects of various shapes.
(2) Capable of handling simultaneously various mechanisms of interactions among discrete objects.
(3) Adding new discrete objects or new algorithms (for detecting contact, determining impact, or determining the motion of objects) into the framework should be easy and does not need modifications on already existing kernel classes.
(4) The simulation procedure can be easily changed.

Constructing a versatile DOSS is indeed more complicated than constructing a monotype one. What makes a versatile DOSS so hard to build? Table 1 compares a versatile DOSS with a monotype DOSS. A system for simulating a single (monotype) material includes only one type of discrete object, and so needs only to model one type of discrete object and its behavior. However, in a versatile DOSS, discrete objects of every type have their own attributes and behavior, and the interactions among different combination of types need different contact-detection and impact-solution algorithms. For example, consider a simulation involving three types of discrete objects, namely, ObjectA, ObjectB, and ObjectC. The contact detection and impact solution are required for the following six ($=3*(3+1)/2$) pairs: ObjectA_ObjectA, ObjectA_ObjectB, ObjectA_ObjectC, ObjectB_ObjectB, ObjectB_ObjectC, and ObjectC_ObjectC. However, it should be noted that the numbers shown in Table 1 indicate the theoretical maximum cost. The actual cost is usually less than that. For example, if ObjectC is a type of boundary object which does not move, there is no need to consider the ObjectC_ObjectC case. Also, if all types of the objects have the same geometric shape, only one algorithm is needed for contact detection. Because the complexity of the interactions among different types of discrete objects for versatile DOSS does exist (as shown in Table 1), VEDO should be designed to be capable of handling the worse case scenario.

**Table 1** Algorithms required by various DOSSs

| DOSS Type | Objects Modeling | Contact Detection | Impact Solution | Objects Motion |
|---|---|---|---|---|
| Monotype DOSS | 1 | 1 | 1 | 1 |
| Three Types DOSS | 3 | 6 | 6 | 3 |
| Versatile (n type) DOSS | n | $n(n+1)/2$ | $n(n+1)/2$ | n |

Besides the requirements for a flexible underlying software framework discussed above, there are other issues to be addressed in development of a complete DOSS package. For example, the simulation usually involves a large number of objects, representing the discrete objects and their interactions. These objects may come into and out of existence many times during the simulation. Object creation can be very time consuming and becomes an issue. The file I/O and rendering (or visualization) of simulation results, especially with very large number of objects, also need to be carefully addressed. However, these issues are more related to implementation strategies and are outside the scope of the present study. Therefore, they are not discussed here, except that some discussions are provided on reducing the amount of object creation for interactions among discrete objects.

## 3 Design of Kernel classes

Figure 2 depicts the core design of VEDO. The design consists of four main classes: *Interaction*, *DiscreteObject*, *ContactDetector,* and *ImpactSolver*. The underlying concept of VEDO is that each pair of *DiscreteObject* is served by one *Interaction* and each *Interaction* holds two algorithms, namely *ContactDetector* and *ImpactSolver*. Each *Interaction* can provide each pair of *DiscreteObject* with detecting the contact and solving the impact forces if each *Interaction* holds appropriate algorithms. It should be noted that the present design does not impose the limitation on the allowable contact points for each contact pair. It is the responsibility of the corresponding *ContactDetector* and *ImpactSolver* classes to implement appropriate algorithms for handling multi-points contact between two types of discrete objects. Furthermore, the design presented here is mainly devoted to the explicit time integration and meets the requirements of DOSS discussed in the previous section that the total effect of all interactions among discrete objects can be regarded as the integration of individual interactions between any two discrete objects.

### 3.1 Kernel classes

*DiscreteObject* is an abstract class that comprises all common attributes of a discrete object, including velocity, mass, position, and others without implementation of its motion and description of its shape. Each concrete subclass (such as *DOShpere* or *DOPlate*) derived from *DiscreteObject* defines the attributes of the particular shape of discrete objects and implements the

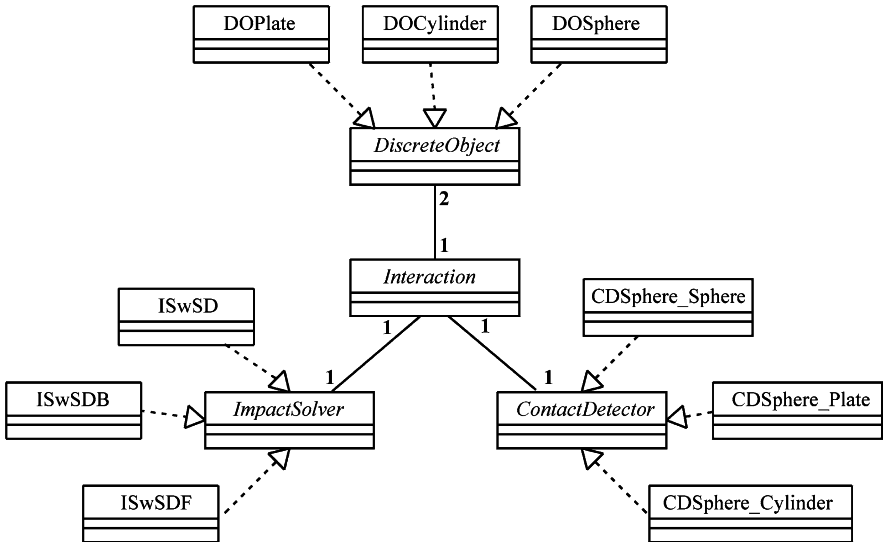



**Fig. 2** Relationships among Kernel Classes

operation for object motion individually. For instance, a subclass that models spherical discrete objects must define radius and implements how it moves under external impacts.

*ContactDetector* and *ImpactSolver* are also abstract classes used to define the interfaces of two groups of algorithms. Each concrete *ContactDetector* subclass implements one contact-detection algorithm and holds information about contact between discrete objects, such as contact point(s) and contact direction. Each *ImpactSolver* subclass implements one algorithm that determines the forces between discrete objects that are in contact.

*Interaction* is designed to link two *DiscreteObject* subclasses to two algorithm subclasses derived from *ContactDetector* and *ImpactSolver*. The two algorithm subclasses simply meet the requirements of modeling the interaction mechanism between two *DiscreteObject* subclasses. For instance, when the interaction between a basketball and a plate is considered, *Interaction* links *DiscreteObject* subclasses (i.e., *DOSphere* and *DOPlate*) to the *CDSphere_Plate* subclass of *ContactDetector* and to the *ISwSDF* subclass of *ImpactSolver* that considers elastic spring, damping, and friction forces when the two objects collide.

The design of *DiscreteObject*, *ContactDetector*, *ImpactSolver* and their subclasses is knows as **Strategy** [5]. These three groups of classes define three families of algorithms - the determination of the motion of discrete objects, the detection of contact between two discrete objects, and the determination of interaction mechanism. The design of *Interaction* is known as **Mediator** [5]. *Interaction* not only links these three families of algorithms but also encapsulates all operations among them.

Different subclasses of *DiscreteObject*, *ContactDetector*, and *ImpactSolver* can be implemented for different applications. Their implementation is simply an addition to the existing framework and does not require modification on any existing classes. Any interaction mechanism between *DiscreteObject* subclasses can be handled by *Interaction* that links the *DiscreteObject* subclasses to appropriate subclasses of *ContactDetector* and *ImpactSolver*. For example, when modeling the interaction of two spherical colloidal particles, all the developers need to implement are three subclasses. They are an *ImpactSolver* subclass that considers van der Waals forces and the electrostatic force, a *ContactDetector* subclass that detects if any of the given pair of colloidal particles falls into the influence area of the other one, and a *DiscreteObject* subclass that determines the motions of the particles. Accordingly, a DOSS based on VEDO can handle simultaneously discrete objects of various shapes and various interaction mechanisms among discrete objects.

### 3.2 Runtime data structure

Consider a basketball, a baseball, and a fixed floor plate as three discrete objects. The elastic spring, damping, and friction forces govern the interaction mechanism between the basketball and the floor, and that between the baseball and the floor. Only the elastic and damping forces are considered in simulating the interaction mechanism between the basketball and the baseball. Figure 3 presents the runtime data structure of the above example. Iact1, Iact2 and Iact3 are three *Interaction* instances that have their own algorithms for contact detection and impact solution between each pair of the three instances of *DiscreteObject* subclasses (basketball, baseball and floor), respectively.

As shown in Fig. 3, the present design provides great flexibility for simultaneously dealing with various types of discrete objects and various types of interaction mechanisms. The trade-off of such a design is that it seems to consume a large amount of run-time memory since the number of interactions is proportional to the square of the number of discrete objects. However, this

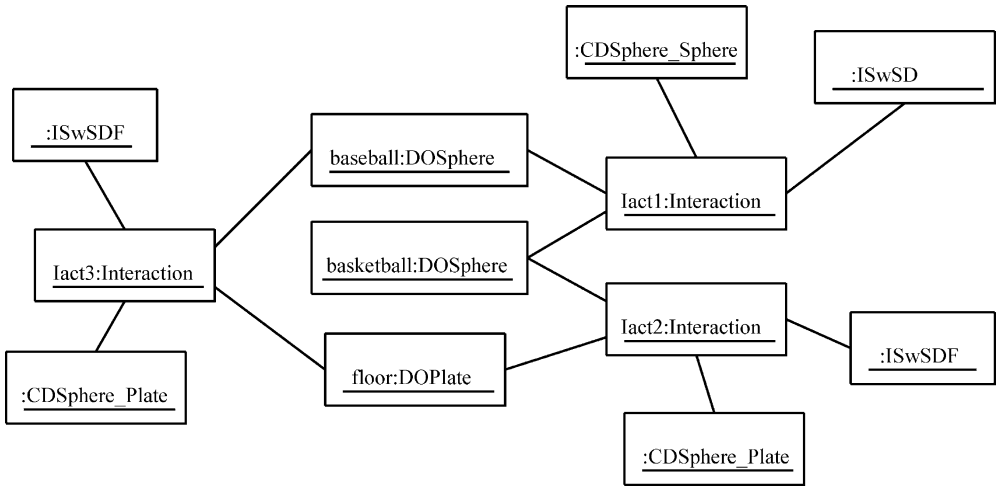



**Fig. 3** Example runtime data structure

issue does not become problematic if some smart strategies are employed. There are at least two implementation techniques for reducing the run-time memory. First, for example, for those pairs of *DiscreteObject* instances that have the same interaction mechanism, their *Interaction* instances can be linked to the same *ImpactSolver* instance to save memory. This is similar to the concept of **Singleton** [5]. Second, some concepts of the fast algorithms discussed in [9] [11] can be used to reduce not only the operations for contact detection among discrete objects but also the number of *Interaction* instances. For example, the interactions can be neglected if two discrete objects are too far away from each other to come into contact within a certain time interval. Nevertheless, the effective use of smart strategies for reducing run time resources and increasing efficiency depends largely on the nature of the application. Different kinds of applications may require implementation of different strategies to achieve efficiency. An example will be presented later in Sect. 5.1 to illustrate how the number of *Interaction* instances can be significantly reduced in the simulation for that particular application. Also, it should be noted that the strategies discussed in this paper are mainly devoted to explicit time integration.

## 4 Design of auxiliary classes

In addition to the kernel classes discussed above, the following three auxiliary functions are needed for automatic and systematic operations in a versatile discrete objects simulation:

(1) Management of simulation information: this function should provide description of the simulation problem and record the simulation result.
(2) System initiation: this function should generate all kernel instances (all instances of the subclasses derived from kernel classes) and link them together appropriately.
(3) Control of simulation procedure: this function should coordinate all kernel instances to complete all simulation rounds.

Five classes, namely *SimMediator*, *DOContainer*, *IactContainer*, *DOWorld*, and *Assembler*, are designed to support the above auxiliary functions (as shown in Fig. 4).

*SimMediator* is a **Mediator** [5] that performs system initiation and carries out the simulation procedure. In

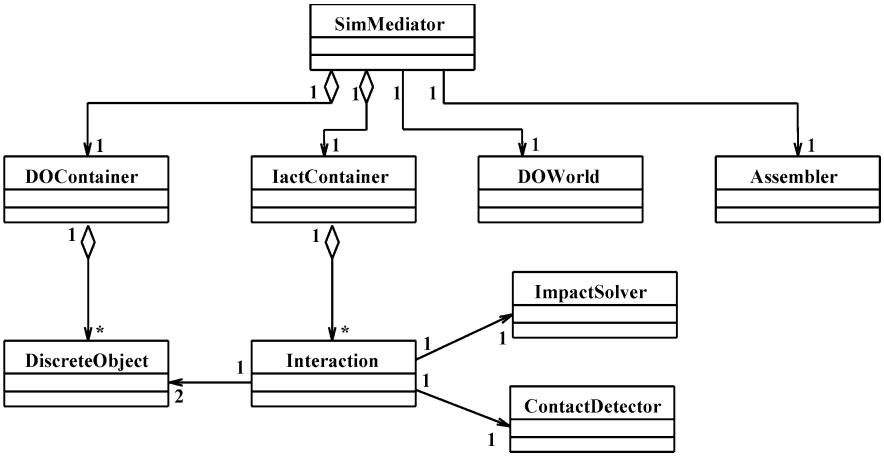



**Fig. 4** Auxiliary classes and Kernel classes

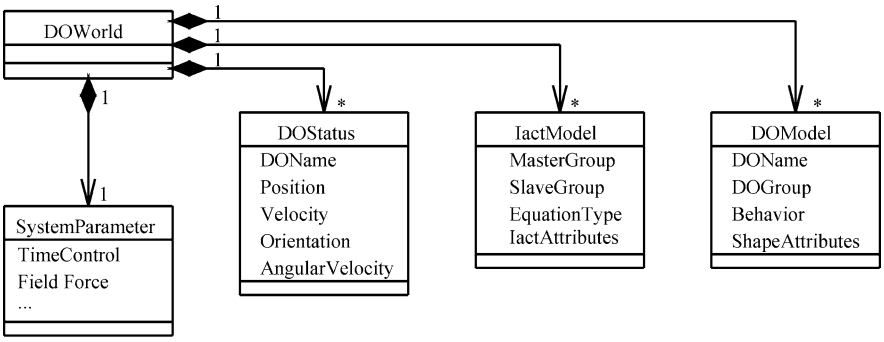



**Fig. 5** Four component classes of *DOWorld*

the system initiation stage, *SimMediator* asks *Assembler* to generate all kernel instances based on the simulation information from *DOWorld*. And, *SimMediator* places all kernel instances in the two specific containers, *DOContainer* and *IactContainer*. *DOContainer* contains all *DiscreteObject* instances and *IactContainer* contains all *Interaction* instances. When all required kernel instances have been generated and put into these two containers, *SimMediator* can drive the simulation through manipulating the instances in the containers. The details of the *DOWorld*, *Assembler*, and *SimMediator* classes are discussed below.

## 4.1 DOWorld

*DOWorld* is responsible for storing simulation information using the classes of four types, namely *SystemParameter*, *DOModel*, *DOStatus*, and *IactModel*, as indicated in Fig. 5. *SystemParameter* stores control parameters and global variables. The control parameters are related to the control of the simulation procedure. They include the time interval of each simulation step, the total number of simulation rounds, the start and end times of the simulation, etc. The global variables contain information global to all discrete objects, e.g., the total number of discrete objects in the simulation and the magnitudes of field forces (e.g., the gravity force) acting on them. *DOModel* stores the model information of a discrete object, including object name, shape attributes, material attributes, behavior (i.e., mobile or fixed), name of the group it belongs to, etc. *DOStatus* stores the status of a discrete object, including the position, orientation, velocity, etc. A complete definition of a discrete object requires the combination of information about its model and its status. *IactModel* stores the information about the interaction mechanism between two given groups of discrete objects. The information includes the physical mechanism governing the motion of each pair of discrete objects (e.g., combination of spring, dashpot, and friction), name of the numerical method used to solve the motion equations, and the values of the attributes of the mechanism, e.g., spring constant, damping ratio, etc.

De-coupling the system information into four parts avoids duplication of information stored and therefore saves memory. For example, discrete objects of the same shape can use the same *DOModel* instance. Different pairs of discrete objects can use the same *IactModel* instance. In addition, it provides good flexibility to modify the interaction mechanism for a pair of discrete objects.

## 4.2 Assembler

*Assembler* is responsible for generating all instances of kernel classes and linking them together appropriately. As indicated in Fig. 6, *Assembler* performs its tasks using four component classes. These are *DOcFactoryDecorator*, *CDcFactoryDecorator*, Is'c*FactoryDecorator*, and *IactBuilder*. *DOcFactoryDecorator* generates all *DiscreteObject* instances based on the model and status information of discrete objects stored in *DOWorld*. *IScFactoryDecorator* generates all *ImpactSolver* instances based on the interaction information in the *DOWorld*. *CDcFactoryDecoratoy* generates all *ContactDetector* instances according to the requirements of each

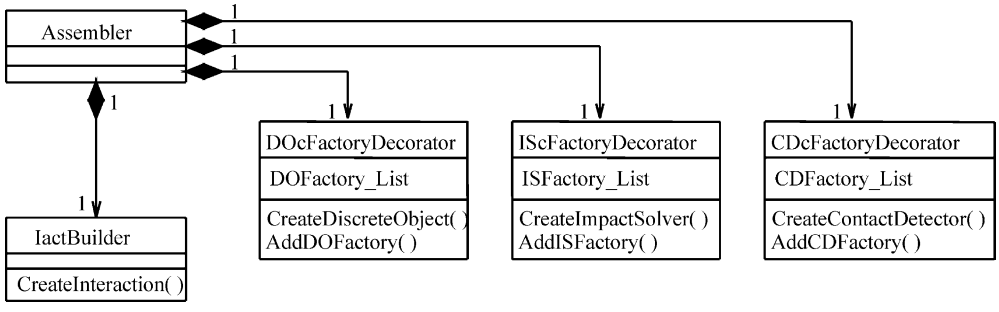



**Fig. 6** Four component classes of *Assembler*

pair of *DiscreteObject* instances. *IactBuilder* generates *Interaction* instances and links four given instances (i.e., two of *DiscreteObject*, one of *ImpactSolver*, and one of *ContactDetector*) to the *Interaction* instance.

Figure 7 presents the detailed design of *DOcFactoryDecorator*. *DOFactory* is an abstract class that defines an interface for generating *DiscreteObject* instances. It lets its subclasses (such as *DOCylinderFactory*, *DOPlateFactory*, and *DOCSphereFactory*) to determine which subclass of *DiscreteObject* to instantiate. Such a design is known as **Factory Method** [5]. This design eliminates the need to bind specific *DiscreteObject* subclasses into the implementation of a DOSS. Instead, the developer needs only to deal with the interface of *DOFactory* for generating *DiscreteObject* instances. However, the design seems to be troublesome because the developer still needs to face very many *DOFactory* subclasses and determine which is suitable for generating the required *DiscreteObject* instances. In this study, *DOcFactoryDecorator* is designed to overcome this trouble. *DOcFactoryDecorator* maintains a list of *DOFactory* instances and provides an operation to append new *DOFactory* instances to this list. Hence, *DOcFactoryDecorator* knows how to generate each user-specified instance of *DiscreteObject* subclass as long as each corresponding instance of *DOFactory* subclass has been appended into the list. *DOcFactoryDecorator* is also a *DOFactory* subclass so the developer needs only to deal with the same interface defined by *DOFactory*. This kind of design is known as **Decorator** [5]. Moreover**,** if the programming language supports generic programming, all *DOFactory* subclasses need simply to be defined as a single generic class because their responsibilities and implementation are similar. For example, all *DOFactory* subclasses can be declared as a single template class in C++ programming language [16]. The design of *IScFactoryDecorator* and *CDcFactoryDecoratoy* follows exactly the same approach as that of *DOcFactoryDecorator*. Although the design of these three component classes may seem to be complicated, their implementation is clean and need not be altered when any new kernel subclass is added to the system. Moreover, the design of *Assembler* and *IactBuilder* that separates the construction process from complex representation of kernel instances is known as **Builder [5]**.

Figure 8 depicts how *Assembler* collaborates with *SimMediator* and other component classes. *Assembler* must respond to two requests from *SimMediator*, that is, generating *DiscreteObject* instances {a} and generating *Interaction* instances {b}. When *Assembler* receives the request for generating a *DiscreteObject* instance from *SimMediator*, it simply forwards this request to *DOcFactoryDecorator* {a.1}. Generating *Interaction* instances is more complicated than generating *DiscreteObject* instances. When *Assembler* receives the request for generating an *Interaction* instance for two given discrete objects, it first asks *IScFactoryDecorator* and *CDcFactoryDecorator* to generate corresponding instances of *ImpactSolver* {b.1} and *ContactDetector* {b.2}. It then asks *IactBuilder* to generate an *Interaction* instance and link the corresponding instances of *DiscreteObject, ImpactSolver,* and *ContactDetector* to the *Interaction* instance {b.3}.

## 4.3 SimMediator

*SimMediator* is responsible for performing system initiation and carrying out the simulation procedure. In practice, different kinds of applications may require different simulation procedures or different kinds of system initiation or both. The developer can implement different control policies (including parallel processing) through derivation of different *SimMediator* subclasses. The collaboration of *SimMediator* with other component classes is examined below for two events, simulation initiation and simulation processing. It should be noted that the cases discussed below are general ones helping to demonstrate the collaborative relationships among all the classes shown in Fig. 4. Smarter strategies that take advantage of the characteristics of the application may be employed to achieve a more efficient implementation.

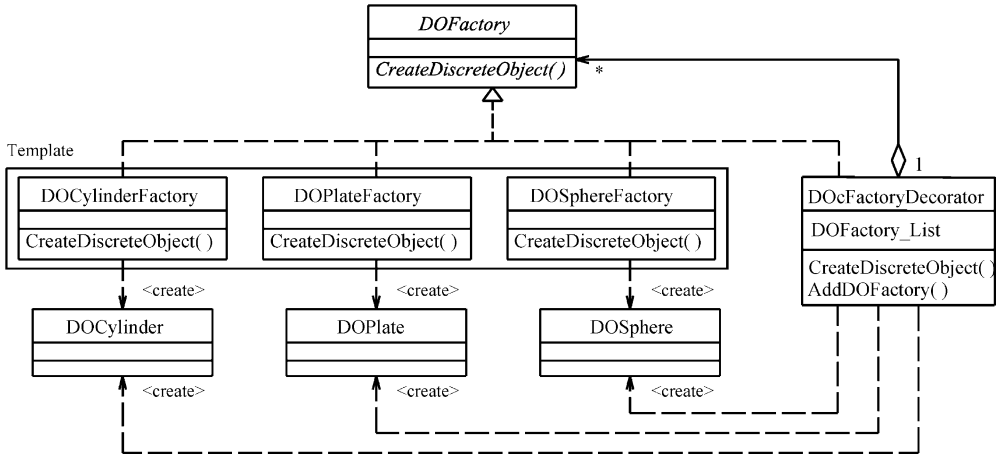



**Fig. 7** Design of *DocFactory Decorator*

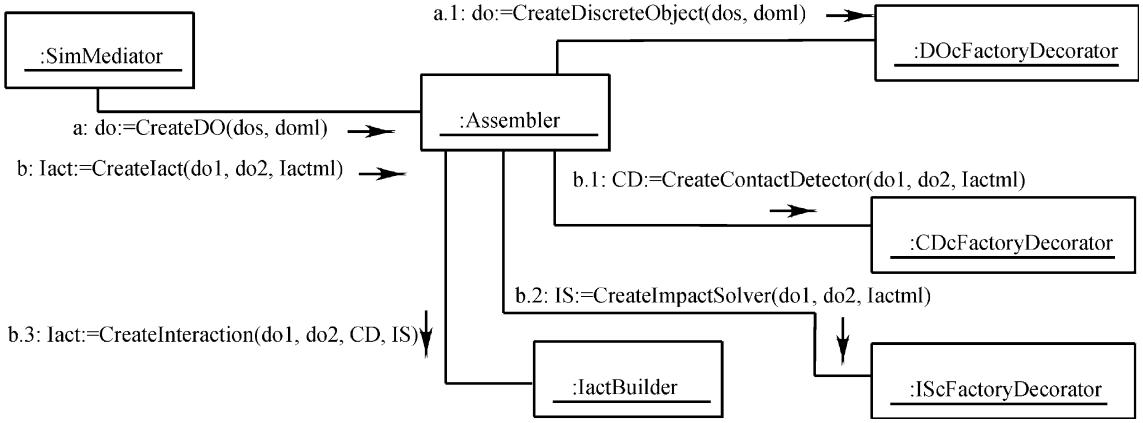



**Fig. 8** Collaboration among *Assembler*, *SimMediator*, and other component classes

### *4.3.1 Simulation initiation*

Figure 9 depicts the collaboration diagram for creating instances of kernel classes at the very beginning of the simulation. Before *Assembler* is requested to generate instances of kernel classes {4 and 8}, *SimMediator* must obtain the information about the number of discrete objects as well as the status and model of each discrete object from *DOWorld* {1, 2 and 3}. The generation of each *Interaction* instance {6.1, 6.2, 7 and 8} depends on the knowledge of the corresponding pair of discrete objects. Accordingly, all *DiscreteObject* instances must be generated before all *Interaction* instances. In addition, each *Interaction* instance is marked as active in the linking stage if the pair of discrete objects it serves comes into contact. Otherwise, it is marked as inactive. After all instances of kernel classes have been created, *SimMediator* puts them into *DOContainer* {5} and *IactContainer* {9}.

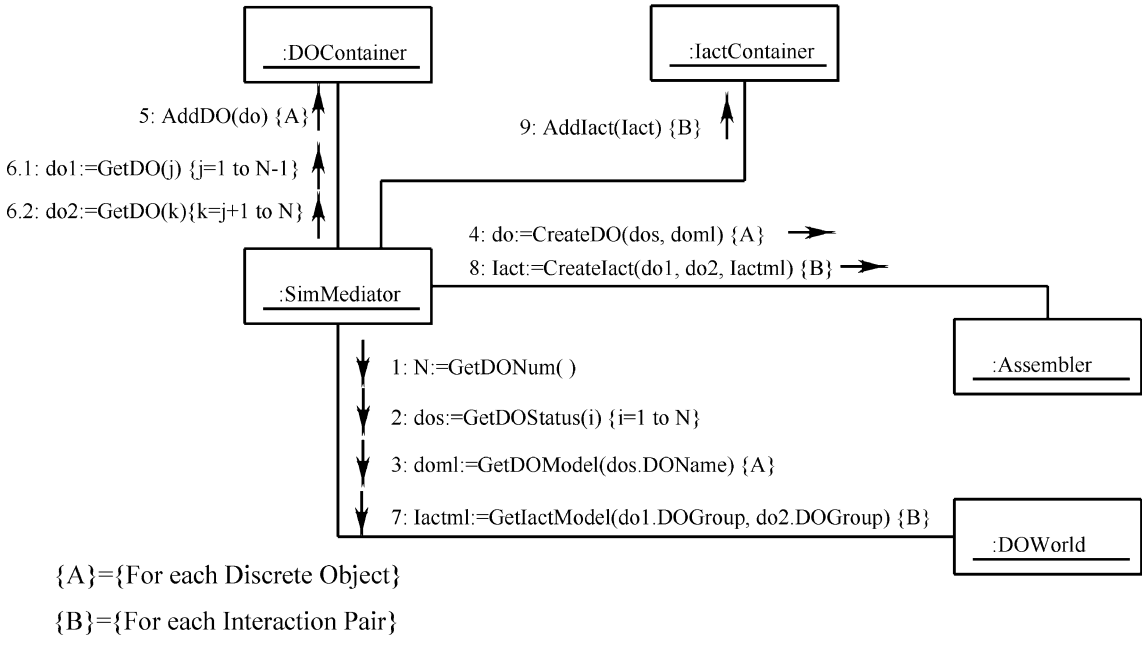
### *4.3.2 Simulation processing*

A discrete objects simulation repeats a certain sequence of tasks during each time interval. It first determines the contact status between any pair of discrete objects. The external impact forces applied to each pair of discrete objects are then calculated according to the contact status of the pair. Finally, each discrete object translates and rotates from its current position and orientation due to the external impact forces.

Figure 10 depicts an example of how *SimMediator* drives one round of the simulation. During a round, *SimMediator* sends out the following requests:

(1) Ask each active *Interaction* instances to calculate external impact forces {1, 1.1 and 1.2}. Each active *Interaction* instance must send the results calculated

**Fig. 9** Collaboration for creating instances of kernel classes



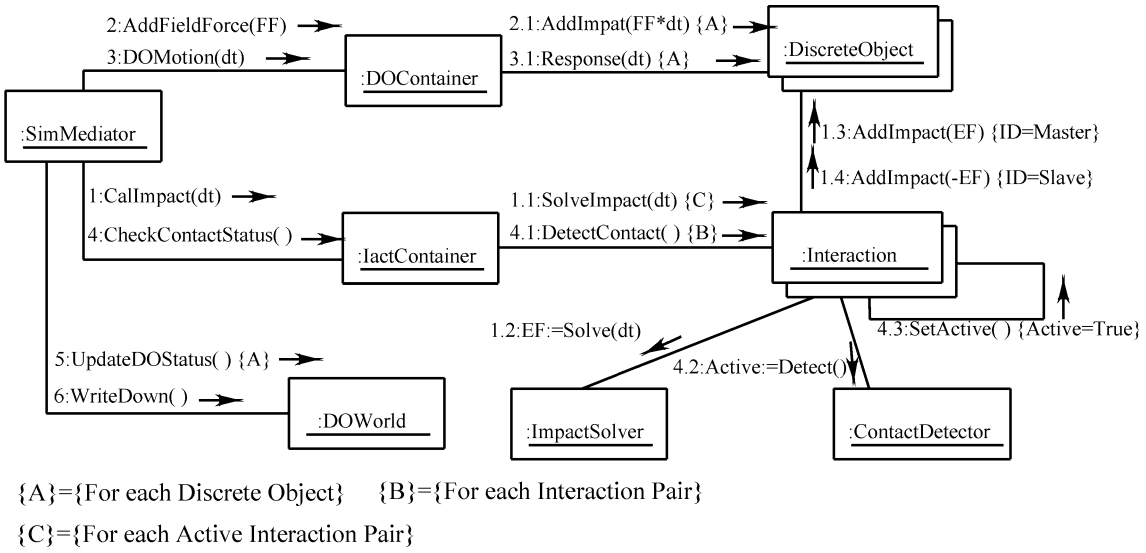



**Fig. 10** Collaboration diagram for simulation processing

by the *ImpactSolver* instance to the two specific *DiscreteObject* instances that it servers {1.3 and 1.4}.

(2) Ask all *DiscreteObject* instances to add the field force through the AddImpact() interface (i.e., the field force is treated like an "impact force" to a DiscreteObject. {2 and 2.1}. In addition, for large scale system where the deviation of the field force is not negligible, *SimMediator* may send different values of the filed force to different *DiscreteObject* instances.

(3) Ask all *DiscreteObject* instances to move {3 and 3.1}.

(4) Ask all *Interaction* instances to check contact status {4 and 4.1}. Contact status is checked to identify the active *Interaction* instances that must be taken care of by *ImpactSolver* instances {4.2 and 4.3}.

After the round is completed, *SimMediator* should update the simulation information in *DOWorld* and request that *DOWorld* write down all simulation information for visualization and analysis {5 and 6}.

## 5 Numerical examples

VEDO has been implemented in the C++ programming language in this study. A discrete objects simulation system, named Knight&Anne, has also been developed on top of the implementation of VEDO. It is used to conduct the studies in the following numerical examples.

### 5.1 A spring mattress example

Consider an example that a red ball (with the radius of 0.8 unit) impacts a spring mattress which consists of 41 x 41 blue balls (with the radius of 0.15 unit) as shown in Fig. 11(a). The red ball is given an initial velocity of 3.0 units/second toward the mattress. Figure 11(b) illustrates the numerical model of this example. Both red and blue balls are modeled by mobile spherical objects and the four boundaries are modeled by fixed cylindrical objects. The interaction among the component objects of the mattress (i.e., the blue balls and the boundaries) is modeled by a two-way spring that can be in either tension or compression. The interaction between the red ball and the mattress is modeled by a one-way spring that only considers compression. Figure 12 is the side view of the simulation process of this example. It shows how the spring mattress deforms to resist the impact of the red ball and finally changes the direction of motion of the red ball.

Figure 13 shows the sequence of displacement contour plots after the impact. The colors on the contour plots present different numerical values of displacements

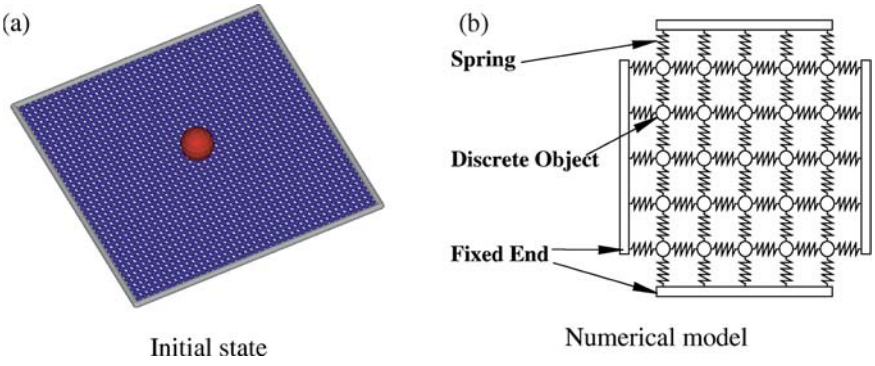



**Fig. 11** Spring mattress example

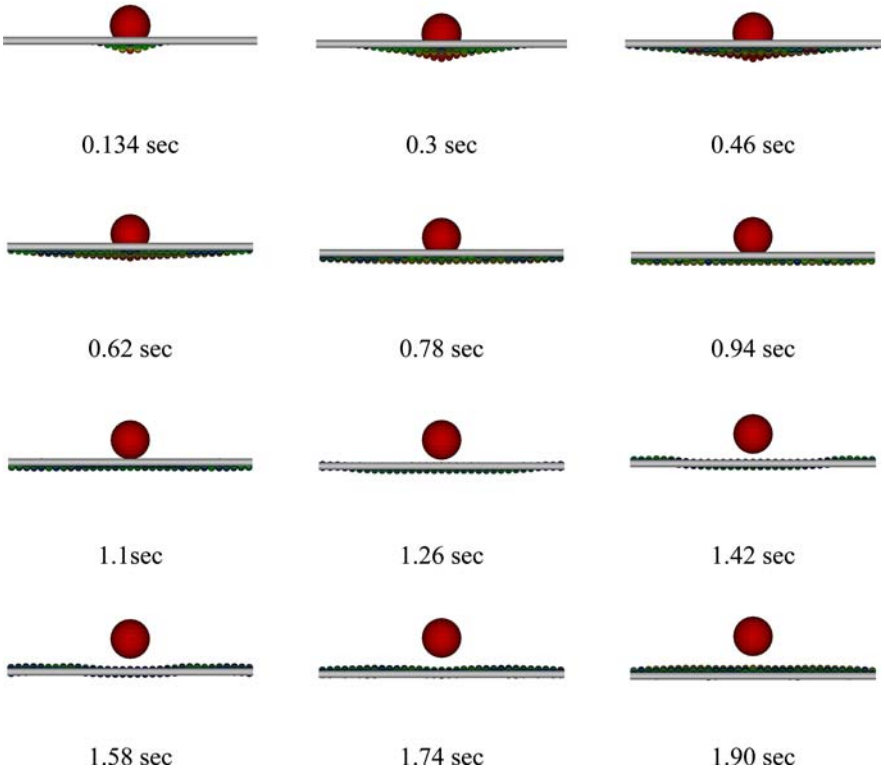



**Fig. 12** Simulation process of the spring mattress example (the Side View)

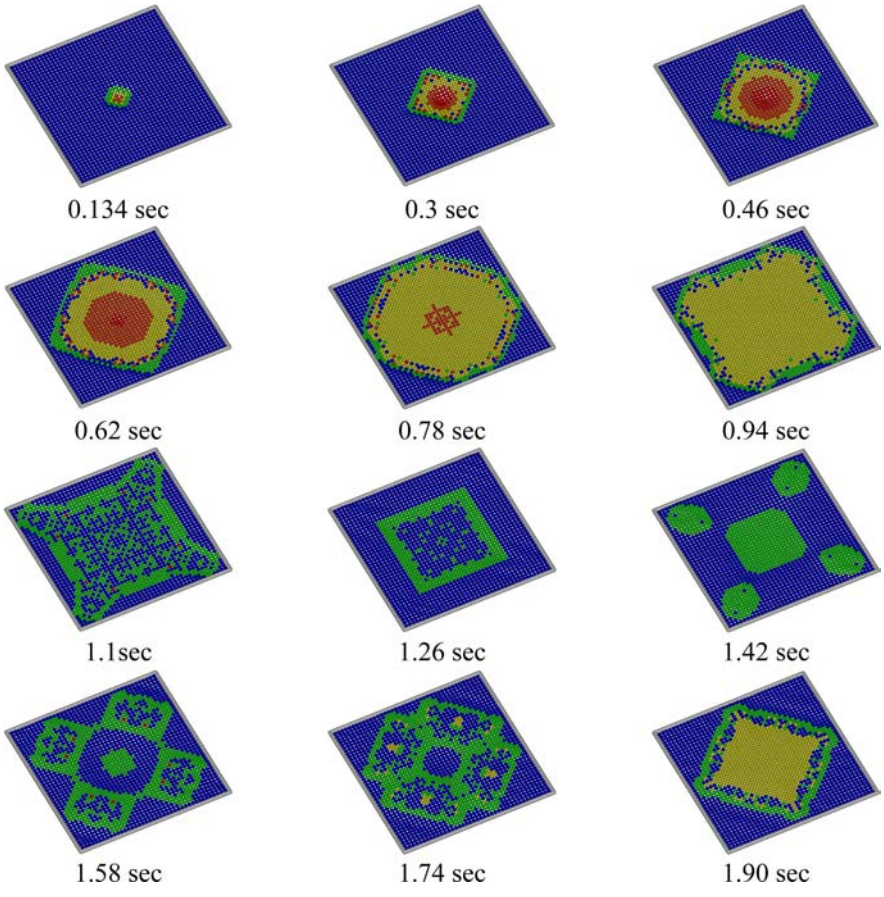



**Fig. 13** The displacement contour plot of the mattress

that are in the descending order of red, yellow, green, and blue. It shows that a wave travels from the center to the fixed end and then reflects from the fixed end.

Based on VEDO, all we have to implement for simulating this example are two subclasses of *DiscreteObject*, two subclasses of *ImpactSolver*, and three subclasses of *ContactDetector*. The two subclasses of *DiscreteObject* models fixed cylindrical and mobile spherical discrete objects. The fixed cylindrical discrete objects do not move no matter how much impact forces are applied on them, while the mobile spherical objects follow the Newton's law to move. The three subclasses of *ContactDetector* are for detecting contacts between (a) fixed-cylinder and blue balls, (b) between a pair of blue balls, and (c) blue balls and the red ball. Although both blue and red balls have the same shape, the interaction mechanism between a pair of blue balls is different from that between the blue and red balls. Therefore, the two subclasses of *ImpactSolver* are for solving the impact forces from the two-way and one-way springs, respectively.

For achieving reduction of CPU time and memory requirements for the simulation, one can implement a *SimMediator* subclass that is capable of filtering out unnecessary *Interaction* instances before asking *Assembler* to generate them. The following characteristics of this example problem can be used to develop a smarter strategy for implementation (see also Fig. 11):

(1) The four boundary objects do not interact with each other and with the red ball.
(2) When the red ball does not collide with the spring mattress, each blue ball only interacts with the same four adjacent discrete objects (which are either blue balls or boundary objects).
(3) When the red ball collides with the spring mattress, only a limited number of blue balls interact with the red ball.

The result of the smarter strategy implemented in this example is a significant reduction of the number of *Interaction* instances (from about 1.4 millions to about 3,700) during the simulation.

## 5.2 Simulation of self-compacting concrete behavior

Some researchers have tried to use discrete objects simulation techniques to study the behavior of (Self-compacting concrete) SCC, e.g., [2][14]. Here let us consider the simulation of the V-funnel and L-box tests that are experimentally available to help understand the characteristics of SCC. A two-phase model is employed to model fresh concrete. In the model, aggregates and mortar are treated as two different types of spherical discrete objects. In the present study, spring, damping, and friction forces are considered between discrete objects. No binding force is considered between the aggregates but a binding force is considered between the aggregates and the mortar. The binding force is a constant force that keeps two discrete objects moving together when they come into contact. A "pull-off" force larger than the binding force is required to separate them. Besides the aggregates and the mortar, two types of discrete objects, fixed plates and fixed cylinders, are used to model the V-funnel and the L-box. Finally, the gravity is also considered as a flied force applying on all discrete objects. Figure 14 is the initial state of the simulation of the V-Funnel test. The red objects model aggregates and the yellow objects model the mortar.

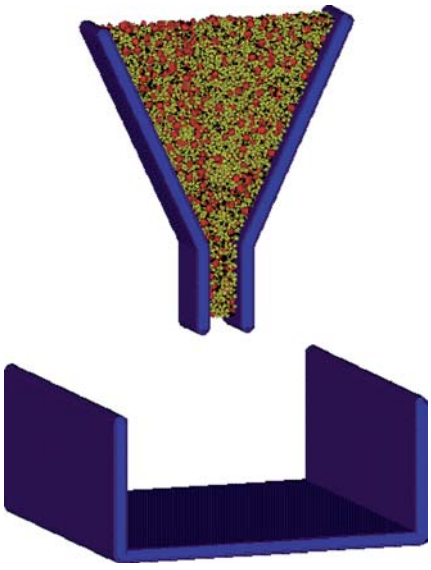


**Fig. 14** Initial state of the simulation of the V-funnel test

Figure 15 shows two simulations with different binding forces that are $10^{-2}$ N and $10^{-3}$ N, respectively. In the case of higher binding forces, it can be observed that the simulated concrete materials have a stickier behavior and cluster together when they are falling down, while the materials have a sandy behavior and leak out continuously in the case of lower binding forces. In addition, the trend observed in the simulations is that the stickier the concrete is, the slower it leaks out. This complies with what has been observed in the experiments.

Figure 16 shows the initial states of two L-box tests. The steel bars are modeled by fixed cylindrical objects. The model for fresh concrete is the same as that in the V-funnel test.

Figure 17 shows four simulations of the L-box test considering different spacing between steel bars and different binding forces. It can be clearly observed that the greater the spacing between the steel bars is, the easier the fresh concrete flows through the bars. In addition, the stickier the fresh concrete is, the slower it flows and the shorter distance it can finally reach.

For simulations in this example, in addition to the implementation efforts done in the previous example, we only need to derive and implement three subclasses from VEDO:

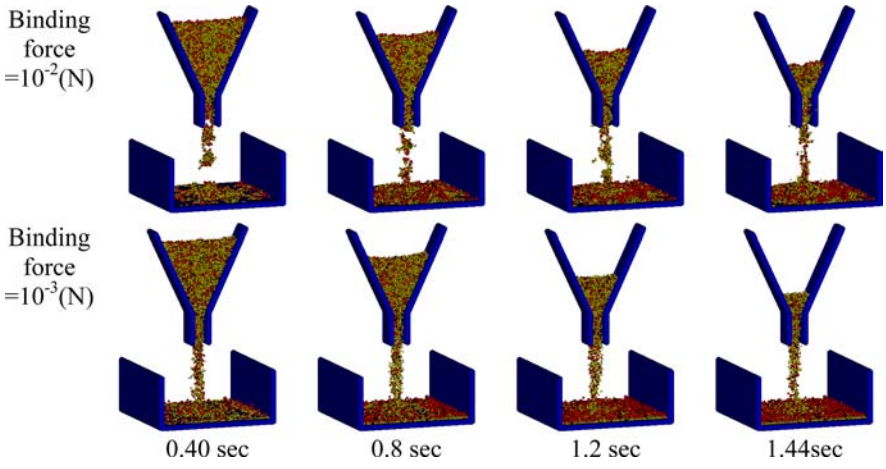



**Fig. 15** Simulation of the V-funnel test

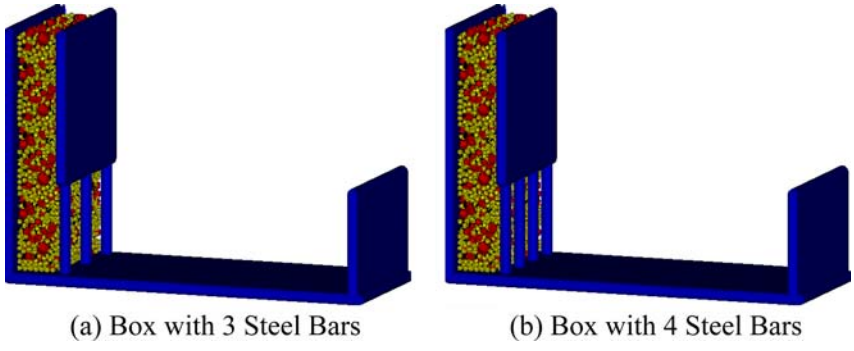



**Fig. 16** Initial states of the simulations of the L-box test

(1) One subclass of *DiscreteObject* for modeling the fixed boundary plate,
(2) One subclass of *ContactDetector* for contact detection between the spherical objects and the boundary plates, and
(3) One subclasses of *ImpactSolver* for handling the binding forces.

### 5.3 Centrifugal casting of colloidal ceramic particles

Colloidal forming techniques have been used to improve the properties, reliability, and functionality of ceramic materials. Particle-packing structures are substantially controlled during the forming process by the colloidal chemical properties of the suspensions, such

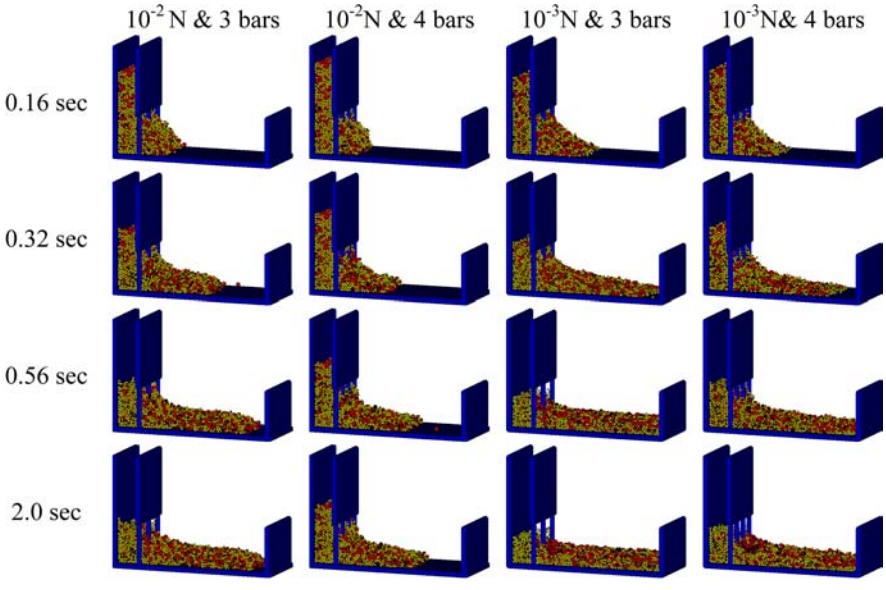



**Fig. 17** Simulation process of L-Box test

as structure of electrical double layer, zeta-potential, Debye-Huckel length, and ionic concentrations [7]. When it comes to microscopic scale, the surface forces of colloidal particles dominate the interactions between particles. At the long range (4 nm to 1 μm approximately), the well-known Derjaguin-Landau-Verway-Overbeek (DLVO) theory containing van der Waals attraction and electrostatic repulsion dominates. According to DLVO theory, the stability of colloids in suspension is determined by the repulsion of double layer electrostatic potential energy and the attraction of van der Waals potential energy as shown in Fig. 18 [15]. To avoid the numerical singularity of DLVO potential energy at the nearly contact range (less than 4 nm approximately to physically contact), the Johnson-Kendall-Roberts (JKR) adhesive interaction model is employed here [8]. When two colloidal particles come into contact, an attractive potential keeps the particles in contact. Consequently, a "pull-off" force is needed to separate them. The governing equation of particle updating is derived from Langevin's equation [4], in which the medium's viscosity is considered. The example studied here is used for studying particle-packing dynamics during centrifugal casting of $Al_2O_3$ colloidal particles. Long-range potential interactions, medium viscosity, gravitational forces, very-short-range adhesion, and solid-body contact are considered. In addition, the results are compared with the simulation done in [7].

For simulation of centrifugal casting, 400 totally dispersed trimodal $Al_2O_3$ particles (80 with the radius of 0.275 μm, 240 with the radius of 0.225 μm, and 80 with the radius of 0.175 μm) are randomly generated in a 12.74 μm × 12.74 μm square box. This arrangement corresponds to an area density of 40%. A centrifugal acceleration of 3,000 $g$ to the left with gravity ($g$) downward is applied as shown in Fig 19.

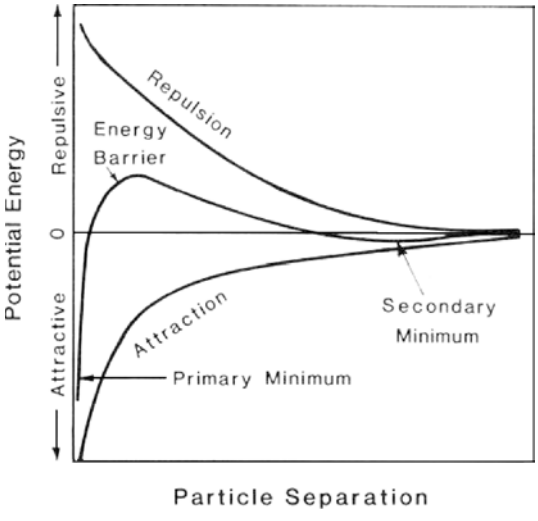



**Fig. 18** Potential energy curve of DLVO theory [15]

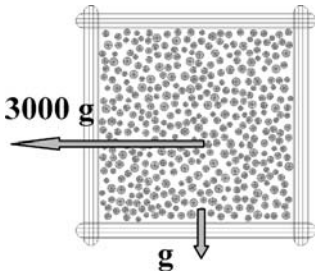



**Fig. 19** Initial state of the colloidal particles simulation examples

Four different suspensions with zeta potentials of 56, 52, 34, and 4 mV are investigated here. Their DLVO interaction forces are shown in Fig. 20. In the case of zeta potential of 56, 52, and 34 mV, the repulsive interaction dominates. However, in the case of zeta potential of 4 mV, the attractive interaction dominates.

Figure 21 shows the successive snapshots from the initial state to the time of 5 *msec* for different zeta potential cases. Chain formation is observed for zeta potentials of 56, 52, and 34 mV, while zeta potential of 4 mV shows more compact packing in which particle agglomeration occurs before they reach the sediment. In addition, the more stabilized the suspension is, the slower they come to sediment. This corresponds with experimental observations, as well as the simulation results of [7].

The particle motion behavior and the interaction mechanisms in the colloidal particles simulations described above are quite different from those in the previous two examples. However, for making the simulations possible, we only need to derive and implement three more subclasses of kernel classes from VEDO, in addition to the implementation efforts of the previous two examples. These three subclasses are

(1) One subclass of *DiscreteObject* for determining the motions of colloidal particles,
(2) One subclass of *ContactDetector* for detecting if any given pair of colloidal particles fall within the influence area of each other, and
(3) One subclass of *ImpactSolver* for solving the potential forces between any given pair of colloidal particles.

## 5.4 Simulation of sieve analysis

Sieve analysis is a common method to obtain the size distribution of aggregates or sands in civil engineering. To model the aggregates in a sieve analysis as polyhedral objects [18], this study has derived and implemented the subclass of *DiscreteObject* for determining the motions of polyhedrons from VEDO. The motion modeling follows the probability approach of [12][18]. In addition, subclasses of *ContactDetector* and *ImpactSolver* are implemented for contact detection and

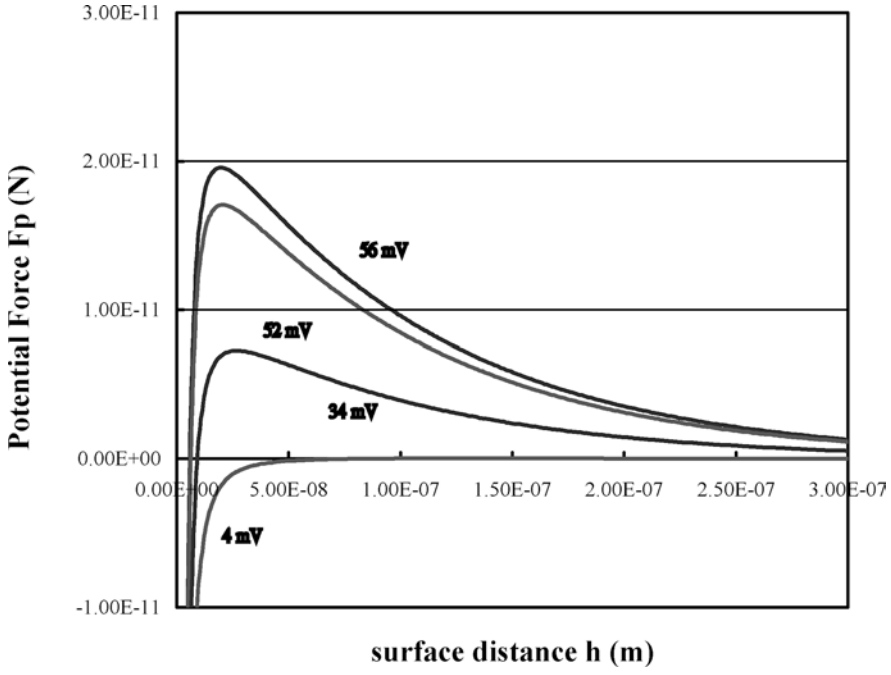



**Fig. 20** Potential forces of different zeta potential

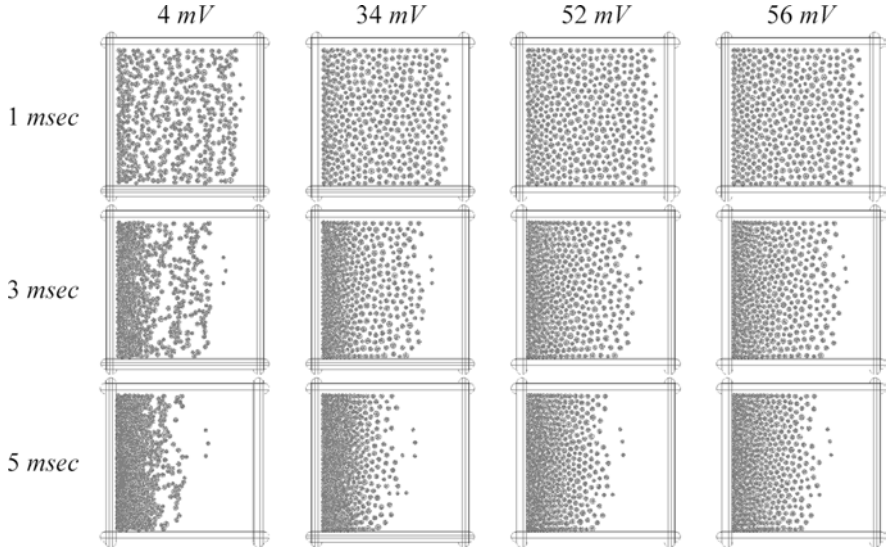



**Fig. 21** Centrifugal casting of colloidal ceramic particles

solution of impact forces, respectively, between a pair of polyhedrons. The algorithms for shape modeling and contact detection of polyhedrons follow the approach of [17] using Linear Programming Algorithm [22]. Figure 22 shows the simulation results of a sieve analysis.

## 6 Conclusions

A versatile framework for discrete objects simulation, called VEDO, has been proposed in this paper. The requirement analysis and object-oriented design of the framework have been presented and discussed. The use of design patterns makes the design of VEDO easier to be understood and reused. It can be seen from the demonstration examples that VEDO has been successfully designed to deal with discrete objects of various shapes and various interaction mechanisms between discrete objects. VEDO is also designed with great flexibility for adding new discrete object shapes and solution algorithms for discrete object interactions. With VEDO, it is believed that the researchers can pay less programming effort to develop a DOSS to meet their specific research needs and, therefore, focus more on the study of domain problems.

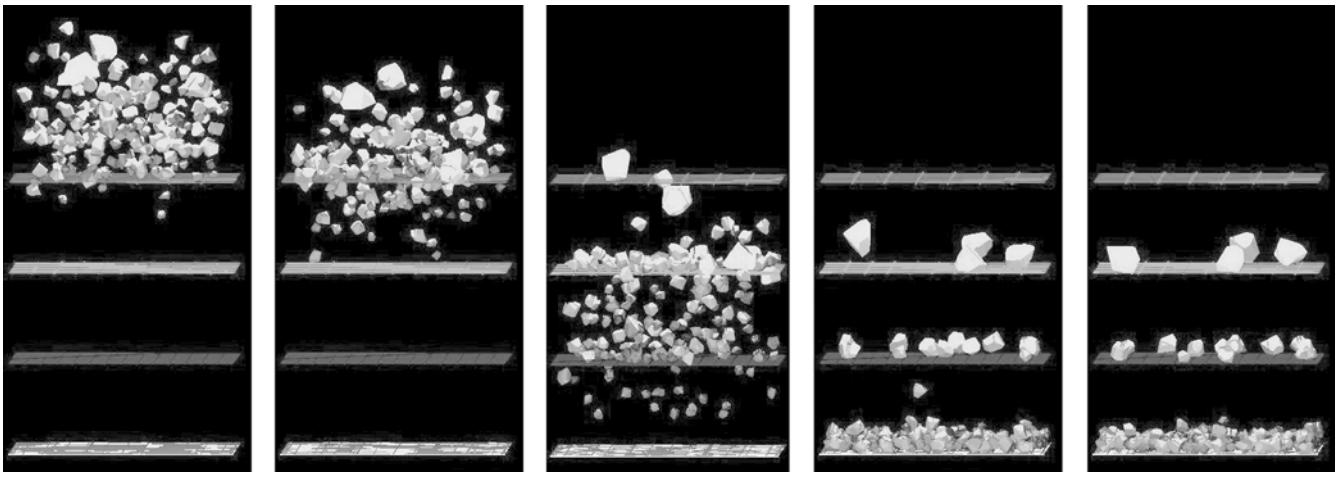


**Fig. 22** Simulation of sieve analysis

## References


1. Booch G, Rumbaugh J, Jacobson I (2001) The Unified Modeling Language User Guide. Addison-Wesley, USA, 8th Printing
2. Chan YW, Liu YS (2003) Parametric Study of Discrete Element Method in Self-Compacting Concrete. Proceedings of 15th KKCNN Symposium on Civil Engineering, Singapore
3. Chiou JD (1998) A Distributed Simulation Environment for Multibody Physics. Ph.D. Dissertation, The Department of Civil and Environmental Engineering, Massachusetts Institute of Technology, MA, USA
4. Ermak DL, Buckholz H (1979) Numerical Integration of the Langevin equation: Monte Carlo simulation. J. Comput. Phys. 35:169–182
5. Gamma E, Helm R, Johnson R, Vlissides J (2002) Design Patterns Elements of Reusable Object-Oriented Software. Addison-Wesley, USA, 23rd Printing.
6. HCItasca, Inc. (2004). PFC$^{3D}$ - Particle Flow Code in 3D. http://www.itascacg.com/pfc.html, HCItasca, Inc., Minneapolis, MN, USA.
7. Hong CW (1997a) New Concept for Simulating Particle Packing in Colloidal Forming Processes. J. Am. Ceram. Soc. 80(10):2517–2524
8. Hong CW (1997b) From Long-Range Interaction to Solid-Body Contact between Colloidal Surfaces during Forming. *J. Euro. Ceram. Soc.* 18:2159–2167
9. Iwai T, Hong CW, Greil P (1999) Fast Particle Pair Detection Algorithms for Particle Simulation. Int. J. Modern Physic C. 10(5):823–837
10. Iwashita K, Oda M (1998) Rolling Resistance at Contacts in Simulation of Shear Band Development by DEM, J. Eng. Mech. 124(3):285–292
11. Kim DJ, Guibas LJ, Shin SY (1998) Fast Collision Detection among Multiple Moving Spheres. IEEE Transaction on Visualization and Computer Graphics, 4(3):230–242
12. Lee Y, Yang CT, Chien CS (2003) A 3D ellipsoid-based model for packing of granular particles. Int. J. Comput Appl. Tech. 17(3):148–155
13. Li JF, Yu WH, Chen CS, Wei WCJ (2003) Modeling Nano-sized Colloidal Particle Interactions with Brownian Dynamics using Discrete Element Method. *Nanotech 2003*, February, San Francisco, USA
14. Noor MA (2000) Three-Dimensional Discrete Element Simulation of Flowable Concrete. Ph.D. Dissertation, The Department of Civil Engineering, Graduate School of the University of Tokyo, Tokyo, Japan
15. Reed JS (1995) Principles of Ceramic Processing 2nd ed., John Wiley & Sons, NY
16. Stroustrup B (2000) The C++ Programming Language, Special Edition. Addison-Wesley, USA, 2nd Printing
17. Yang CT (1999) Simulation System for Three Dimensional Particle Packing, Master Thesis, Department of Civil Engineering, National Cheng Kung University, Taiwan, ROC (in Chinese)
18. Yang CT, Hsieh SH (2001) A Probability-Based Discrete Element Method for Simulation of Sieve Analyses, Proceedings of 14th KKNN Seminar on Civil Engineering, Kyoto, Japan
19. Williams JR, Lim D, Gupta A (1992) Software Design of Object Oriented Discrete Element Systems, Proceedings of Third International Conference on Computational Plasticity, Barcelona, Spain, April 6–10 1992
20. Bernhard P, Algis D (2002) Numerical Simulation of the Motion of Granular Material using Object-Oriented Techniques. Comput. Meth. Appl. Mech. Eng. 191:1983–2007
21. Blilie C (2002) Patterns in Scientific Software: An Introduction. Comput. Sci. Eng. 48–53
22. Murty KG (1983) Linear Programming, John Wiley & Sons, USA