Building Simulation 2009

Eleventh International IBPSA Conference
Glasgow, Scotland
July 27-30, 2009

METHODOLOGY FOR DEVELOPING REUSABLE SCHEDULER CLASSES APPLICABLE FOR LONG TERM BUILDING ENERGY SIMULATION

Eisuke Togashi and Shin-ichi Tanabe
Waseda University, Dept. of Architecture, 3-4-1 Okubo Shinjuku-ku, Tokyo 169-8555
E-mail : e.togashi@gmail.com   http://gf.hvacsimulator.net

ABSTRACT

The aim of this study is to accelerate developments of building simulation programs by using Object-Oriented programming. A reusable generalised scheduler classes and interfaces [1] for defining schedules in simulation programs were developed. *ITermStructure*[2] the "interface" for a term structure was developed to make complex term structure general. By using a "Composite-Pattern", all the concrete term classes that implements *ITermStructure* could be integrated into a complex tree structure. By using the "Type parameter", specifications of a schedule class should not be specified until a scheduler classes is declared and instantiated[3] by a client code. Since the program codes developed in this research and a code made by a client were separated clearly, developed scheduler classes were generally applicable for long-term simulation programs. A concrete example of program, which uses the developed scheduler classes to control cooling tower's operating schedules, was given. It demonstrates that the developed scheduler classes worked fine without any modifications by clients.

INTRODUCTION

A time scheduling is an indispensable function to execute a simulation program, which has time concept such as an annual building energy-calculating program or a daily heat-load calculating program.

As schedules have a large impact on the result of simulation, we should configure them carefully. However, many existing simulation software controls schedules with a simple structure like using arrays of numeric type object. Therefore, they cannot treat arbitrary type of schedules, which could be more complex and be constructed from more than one data type. It costs too much if we make a specific scheduler routine every time when new simulation software is developed. However, it is not possible to unify them, since different simulation software needs a different content of schedules.

Recent object oriented programming (OOP) theories and practices might solve this problem. OOP allows making a program abstractly to expand the versatility of the program. Using OOP, a "concrete class" of schedule is not necessary until a scheduler is declared and instantiated by a client code.

The aim of this study is to present the method to design the scheduler classes with OOP, which can be generally applicable for various simulation programs without modifying source code.

There have been several novel works, which try to develop probabilistic model for occupants' behaviour schedules. For example, J. Page et al. developed an algorithm for the simulation of occupant presence by considering it as an inhomogeneous Markov chain (2008). J. Tanimoto et al. also developed probabilistic model for inhabitants' behaviour schedules and validated with actual measurements (2008). These studies put emphasis on how the occupants' behaviour schedules should be represented stochastically, but they did not mention how to manage these schedules in the programs. The aim of our study is not developing concrete schedules, but to develop the scheduler that can manage various types of schedules.

MERITS OF SOURCE CODE RE-USE

The power of object-oriented systems lies in their promise of code reuse. The code reuse has several advantages. The reuse of program components speeds up software reliability and maintainability. It also shortens the development period for programs.

Doi et al. analyzes 2732 Java programs to evaluate relationships between class-reuse and maintainability of programs (2005). They perform statistical analysis concerning impacts of class-reuse on source code changing in version-upgrades of Java software, and observe strong correlation.

Monden et al. quantitatively analyzed the relation between code clones (duplicated code section) and the software reliability of twenty years old software (2002). As a result, they found that modules having code clones are 40% more reliable than modules having no code clone on average.

[1] An interface is the collection of methods and fields that a class permits objects of other classes to access.
[2] Reserved programming keywords and names of interface, class and method are written in italic type in this paper.
[3] The instance is the actual object created at runtime.

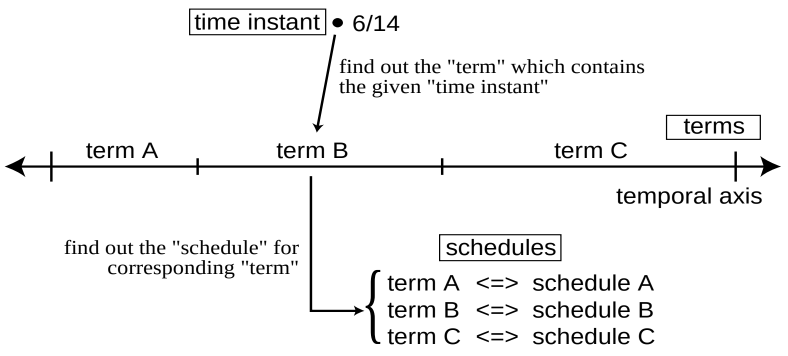


Fig. 1 Functions of scheduler

Change, when requested late in a software development project, can be more than an order-of-magnitude more expensive than the same change requested early (R. Pressman, 1982). If the scheduler can be applicable without source code modification, it has large cost reduction effect in software development, since it is an indispensable function for any simulation programs.

METHOD TO ACHIVE SOFTWARE REUSABILITY

Scheduler's primary function is to make a correlation between a "schedule" and a "time instant". This function can be achieved with two-step approach. First, find out the "term" which contains the given "time-instant". Second, find out the "schedule" for corresponding "term". Figure 1 shows functions of a scheduler. The scheduler is chiefly composed of three components, "time-instant", "term" and "schedule". Generality of scheduler program depends on how scheduler represents these three components. The more abstract these three components represented, the more general scheduler program becomes. In this study, they were abstracted in the following way.

Abstraction of the "time-instant"

Modern programming languages normally provide a function to represent time instant generally. "*DateTime*" class in java and C#, "Date" class in Ruby and java script, and so on. These built-in functions should be used instead of making a new original time-instant class.

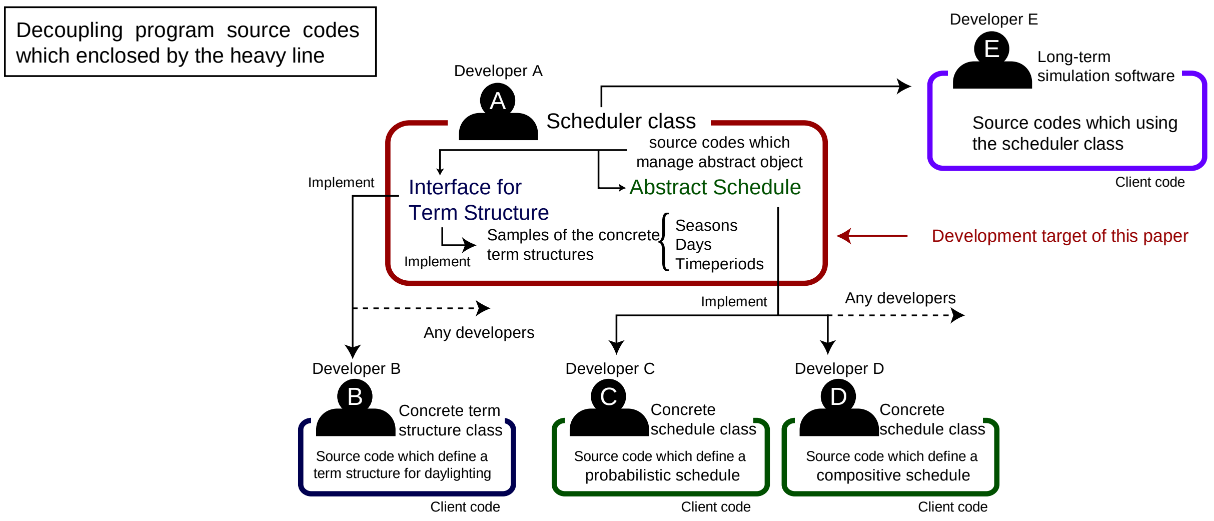
Abstraction of the "schedule"

Many existing simulation programs control schedules with simple structure like using series of numeric type object. For example, schedules are given as single precision boundary variables in the case of the HVACSIM+, the dynamic simulation program. Since a schedule could be more complex and be constructed from more than one data type, in this study, the "type parameters" is used to control a schedule instead.

Abstraction of the "term"

A term is a segment of time. It can be defined by dividing whole time with a certain criterion. For example, time can divide into 7 terms according to the day of the week. In the same way, time can divide into 2 terms (daytime and nighttime) based on a state of sun. A time dividing criterion makes a set of terms. Hereinafter we call these set of terms and the time dividing criterion "term structure".

As discussed above, scheduler needs to know correspondence relationship between the "term" and the "time-instant". For the scheduler, it does not matter how the term structure divide time. Therefore,



Decoupling program source codes which enclosed by the heavy line
Developer A
Scheduler class
source codes which manage abstract object
Implement
Interface for Term Structure
Abstract Schedule
Samples of the concrete term structures
Implement
Seasons
Days
Timeperiods
Developer E
Long-term simulation software
Source codes which using the scheduler class
Client code
Development target of this paper
Implement
Any developers
Any developers
Developer B
Concrete term structure class
Source code which define a term structure for daylighting
Client code
Developer C
Concrete schedule class
Source code which define a probabilistic schedule
Client code
Developer D
Concrete schedule class
Source code which define a compositive schedule
Client code

Fig. 2 The functions of the scheduler program and the development target of this research paper

we define the *ITermStructure* interface (the interface for term structure) to separate a term and time-instant mapping function from a time dividing function.

WHOLE PICTURE OF SCHEDULER CLASSES

Figure 2 shows the functions of the scheduler program and development target of this study. The interface for term structure and the abstract schedule are defined in this study. In addition, some concrete term structure classes (*Seasons*, *Days* and *TimePeriods*) are defined as an example. Because the scheduler class manages schedules by abstract term structure and abstract schedule, the scheduler class is structurally decoupled from concrete term structure classes and concrete schedule classes.

Figure 3 shows UML class diagrams. We use the C# programming language as the language of instruction. *ITermStructure* is the interface and the other blocks shows classes. The *Scheduler* class has an instance of *ITermStructure* interface to make a correlation between a schedule and a term. Three concrete classes, the *Seasons*, the *Days* and the *TimePeriods* implements *ITermStructure* interface.

DEVELOPMENT OF *ITERMSTRUCTURE* INTEFACE

Figure 4 shows the source of *ITermStructure* interface. Two public methods, *GetTermNames* and *GetTermName*, were defined. The former method returns a list of term names. The latter method returns a single term name, which correspond to given time instant. The time instant is given as the *DateTime* object. A concrete class that implements the *ITermStructure* interface can provide a function to divide time in its own way. Three examples of concrete term structure classes are shown below.

Seasons class

Seasons class divides whole time into a set of seasons basing on a date. Table 1 shows some public methods of the *Seasons* class. By *AddSeason* method, with using a name of season and a season starting date, a new season is defined. Figure 5 shows an example of *Seasons* class object. Whole year was divided into four parts by date (3/12, 6/20 and 10/5), and each part has season name. Since the second part and the fourth part of the seasons has the same name (seasons B), they were treated as same season. As a result, there were three seasons, season A, season B and season C, in this seasons object.

Since *Seasons* class implements *ITermStructure* interface, an instance of *Seasons* class could be treated as *ITermStructure* object.

Figure 6 shows three seasons, an example of *Seasons* class instance. Three seasons, “Summer”, “Winter” and “Middle season”, were defined. In this case, if a client call *GetTermName* method, the method defined in *ITermStructure* interface, with *DateTime* object

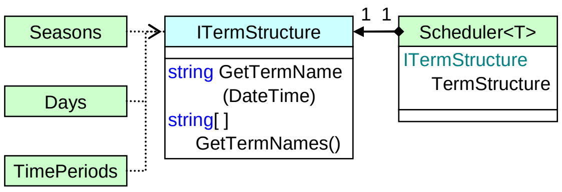


Fig. 3 UML class diagrams

```
/// <remarks>An interface of abstract term</remarks>
public interface ITermStructure {
   /// <summary>Get ID of ITermStructure</summary>
   int ID { get; }

   /// <summary>Get name of ITermStructure</summary>
   string Name { get; }

   /// <summary>Get name list of terms</summary>
   /// <returns>Name list of terms </returns>
   string[] GetTermNames();

   /// <summary>Get term name from DateTime</summary>
   /// <param name="dateTime">Date and time</param>
   /// <returns>Name of term</returns>
   string GetTermName(DateTime dateTime);
}
```

Fig. 4 The source of the ITermStructure interface

Table 1 The public methods of Seasons class

public bool AddSeason(string seasonName, DateTime seasonStartDTime)
=> Divide time by seasonStartDTime and insert new season named “seasonName” to list.
public bool RemoveSeason(int seasonIndex)
=> Remove the season from the list using the index number.
public void GetSeason(int seasonIndex, out string seasonName, out DateTime seasonStartDTime, out DateTime seasonEndDTime)
=> Get the information of the season using the index number.

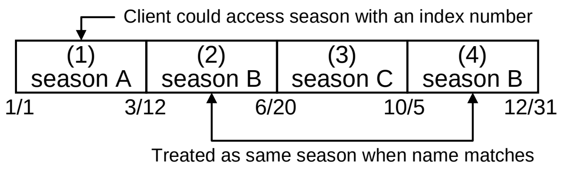

Client could access season with an index number
(1) season A
(2) season B
(3) season C
(4) season B
1/1
3/12
6/20
10/5
12/31
Treated as same season when name matches

Fig. 5 An example of Seasons class object

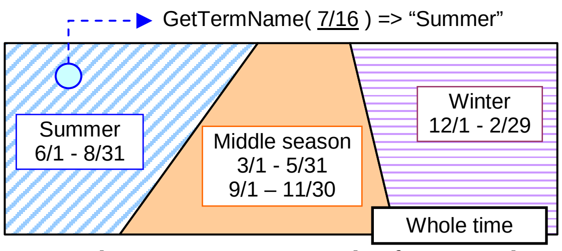

GetTermName(7/16) => “Summer”
Summer 6/1 - 8/31
Middle season 3/1 - 5/31 9/1 – 11/30
Winter 12/1 - 2/29
Whole time

Fig. 6 Three seasons, an example of Seasons class

Table 2 The public methods of Days class

public void ChangeTermName(DayOfWeek dayOfWeek, string termName)
=> Set name of term to day of week.
Public string GetTermName(DayOfWeek dayOfWeek)
=> Get name of term which related to day of week.
Public DayOfWeek[] GetDays(string termName)
=> Get lists of term names.

that represent July 16th as an argument, "Summer" the season name will be returned. If a client call *GetTermNames* method with no argument, 3 dimensions string array filled with "Summer", "Winter" and "Middle season" will be returned.

Days class

Days class divides whole time into a set of days basing on a day of the week names. Table 2 shows some public methods of the *Days* class [4]. Although concrete methods are not same as those of *Seasons* class, both *Seasons* class and *Days* class can be treated as *ITermStructure* object. A specific example will be shown in later section.

TimePeriods class

TimePeriods class divides whole time into a set of seasons basing on a time of day. Since the methods are similar to those of *Seasons* class, they shall not be described here in detail.

DEVELOPMENT OF *SCHEDULER* CLASS

Scheduler class makes a correlation between a schedule and a name of term, which would be defined by *ITermStructure* object. Figure 7 shows the extract source codes of *Scheduler* class. The instance variables were defined in line 6 to 17. The instance methods were defined in line 19 to 53. A content of schedule was controlled by using the type parameters to increase generality of program. A composite pattern was used to represent a complex term structure as simple tree structure. Detailed description of the source codes are shown below.

Abstraction of schedule with using the "Type parameters"

The type parameters make it possible to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated by client code. The concept of type parameters was supported by C# in version 2.0 as "Generics".

Figure 8 is a sample program, which shows how to use type parameters in C#. Client code can use for type arguments when it instantiates Generics class. By using a generic type parameter *T*, client code can use without incurring the cost or risk of runtime casts or boxing operations

In the context of scheduler class, a content of schedule should be represented by the type parameters. Since there could be a various kinds of a schedule according to the client's request, concrete schedule class cannot be determined. In the proposed *Scheduler* class, type parameter *T* is used to create instance (line 9 to 17) and to define public methods (line 19 to 46). Therefore, proposed *Scheduler* class could treat any schedule classes made and provided by clients.

```
 1 ///<summary>Scheduler class</summary>
 2 ///<typeparam name="T">Type of schedule</typeparam>
 3 public class Scheduler<T> : ICloneable
 4 where T : ICloneable
 5 {
 6    ///<summary>Term such as seasons or days</summary>
 7    private ITermStructure terms;
 8
 9    ///<summary>Collections which maps term name to
10                              content of schedule</summary>
11    private Dictionary<string, T> schedules =
12                               new Dictionary<string, T >();
13
14    ///<summary>Collections which maps term name to
15                              child scheduler</summary>
16    private Dictionary<string, Scheduler< T>> schedulers =
17                 new Dictionary<string, Scheduler< T>>();
18
19    ///<summary>Set content of schedule</summary>
20    ///<param name="dateTime">Date and time</param>
21    ///<param name="schedule">Content of schedule</param>
22    public void SetSchedule(DateTime dateTime, T schedule){
23       string sName = getTermName(dateTime);
24       //If child scheduler exists, delegate to child
25       if (schedulers.ContainsKey(sName))
26          schedulers[sName].SetSchedule(dateTime, schedule);
27       //If no child, set content of schedule directly
28       if (schedules.ContainsKey(sName))
29          schedules[sName] = schedule;
30       else schedules.Add(sName, schedule);
31    }
32
33    ///<summary>Get content of schedule </summary>
34    ///<param name="dateTime">Date and time </param>
35    ///<param name="schedule">Content of schedule</param>
36    public void GetSchedule(DateTime dateTime,
37                                     out T schedule){
38       string sName = getTermName(dateTime);
39       // If child scheduler exists, delegate to child
40       if (schedulers.ContainsKey(sName))
41    schedulers[sName].GetSchedule (dateTime, out schedule);
42       // If no child, get content of schedule directly
43       else if (schedules.ContainsKey(sName))
44          schedule = schedules[sName];
45       else schedule = defaultValue;
46    }
47
48    ///<summary>Get name of term from DateTime</summary>
49    ///<param name="dateTime">date and tIme</param>
50    ///<returns>Name of term</returns>
51    private string getTermName(DateTime dateTime){
52       return terms.GetTermName(dateTime);
53    }
54 }
```

Fig. 7 The extract source codes of Scheduler class

```
public class GenericsTest<T> {   //Define type parameter T
   //Declare a variable of type T
   private T gValue;
   //Define property of type T
   public T GenericsValue {
      set{
         gValue = value;
      }
      get{
         return gValue;
      }
   }
}
```

Fig. 8 Sample program, which shows how to use type parameters in C#

[4] The entire source codes mentioned in this paper were released under the General Public License, and could be downloaded from our website.

Building complex segment of time with using "Composite pattern"

The composite pattern is a one of the design patterns proposed by Gang of Four (Erich Gamma et al., 1994). It is used when creating hierarchical object models. The composite pattern defines a manner in which to design recursive tree structures of objects, and the individual objects and groups can be accessed in the same manner.

Figure 9 shows UML class diagrams of Composite pattern. The *Composite* class has method to add and remove child components, and each of those components could also be a composite containing its own children. When the *Operation* method is called, component find a proper child from the child list and call its *Operation* method recursively.

The *SetSchedule* method and *GetSchedule* method of *Scheduler* class (line 22 and 36 in Figure 7) corresponds to the *Operation* method in the Figure 9. If the child *Scheduler* class who can respond to the request exists, parent *Scheduler* class delegate their actual operation to the child class.

Since each *Scheduler* class contains *ITermStructure* interface, hierarchical *Scheduler* classes can represent a compositive term structure. Figure 10 shows an example of compositive term structure. The hatched square is the union of sets "Winter" and "Night time". To represent this union with *Scheduler* classes, select the "Night time" node in a *Days* object and connect the *Days* object to the "Winter" node in a *Seasons* object (Figure 11). This compositive term could be used to schedule a building thermal mass storage in winter.

The child schedulers and schedules were related to term names with hash tables in *Scheduler* class (line 11 and 16 in Figure 7). If there exists a child *Scheduler* object related to the given term name, parent *Scheduler* object calls child method recursively. In this way, client code can manage a schedule with only date and time information.

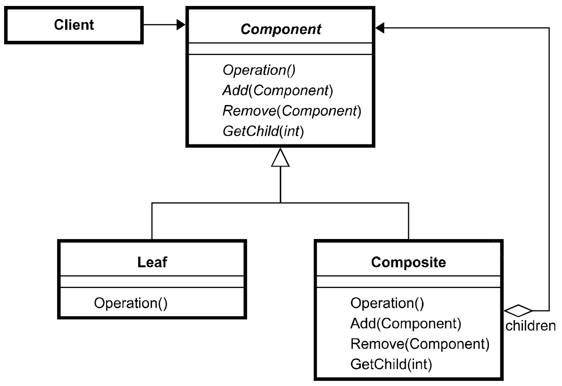


Fig. 9 UML class diagrams of Composite pattern

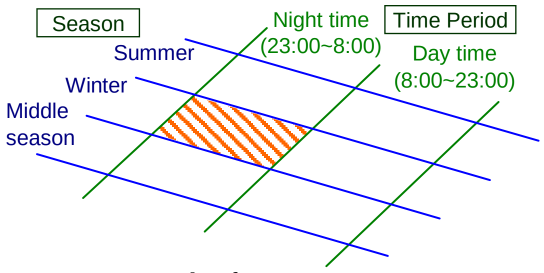
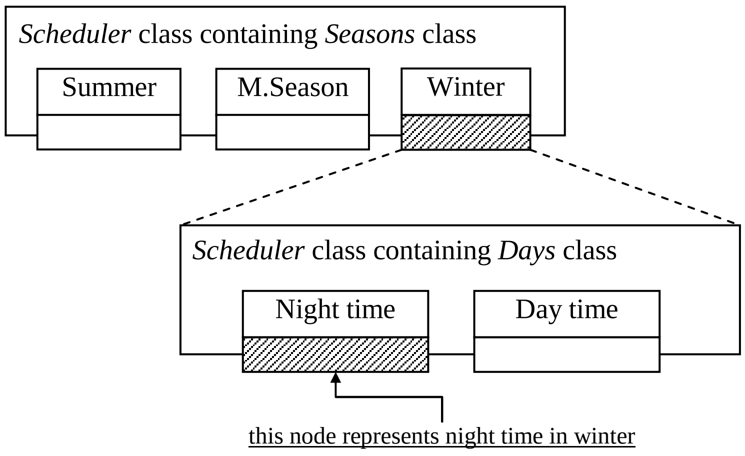


Fig. 10 An example of compositive term structure

SAMPLE OF SCHEDULE CONTROL WITH SCHEDULER CLASSES

In this section, a sample program that demonstrates how to use the proposed *Scheduler* class is shown. A target of scheduling is an operation mode of a cooling tower.

Figure 12 shows a control schedule of a cooling tower. Three operating modes, "No free cooling", "Switch mode with a wet-bulb temperature" and "Switch mode with a dry-bulb temperature" were defined by using enumerated type (line 4 to 11 in Figure 12). Some instance variables were defined from line 13 to 20 to characterize the operating state of the cooling tower. A simple public method to get cooling water outlet temperature was also defined (line 25 to 39). Since the *CTSchedule* class is a complex class who has the enumerated type, the instance variables and the public method, it is



Scheduler class containing Seasons class
Summer
M.Season
Winter
Scheduler class containing Days class
Night time
Day time
this node represents night time in winter

Fig. 11 Representation of compositive term structure with Scheduler classes

```
///<summary>Cooling tower control schedule</summary>
public class CTSchedule {
  /// <summary>Selectable mode</summary>
  public enum Mode {
    /// <summary>No F.C. (Free cooling)</summary>
    NoFreeCooling = 0,
    /// <summary>Start F.C. with wet bulb temp. </summary>
    SwitchWithWBTemperature = 1,
    /// <summary>Start F.C. with dry bulb temp. </summary>
    SwitchWithDBTemperature = 2
  }

  /// <summary>Oparating mode</summary>
  public Mode OperatingMode = Mode.NoFreeCooling;
  /// <summary>Set point of outlet water temp.[C]</summary>
  public double OutletWaterTemperature = 32d;
  /// <summary>S.P. of outlet water temp. in F.C.</summary>
  public double OutletWaterTemperatureFC = 16d;
  /// <summary>F.C. starting temp. [C]</summary>
  public double FCStartTemperature = 15d;

  /// <summary>Get S.P. of outlet water temp.</summary>
  /// <param name="airState">outdoor air state</param>
  /// <returns> S.P. of outlet water temp.</returns>
  public double GetCoolingWaterTemperature(
                                           MoistAir airState) {
    //Case 1 : Start F.C. with dry bulb temperature
    if(OperatingMode == Mode.SwitchWithDBTemperature &&
      airState.DryBulbTemperature < FCStartTemperature)
        return OutletWaterTemperatureFC;

    //Case 2 : Start F.C. with wet bulb temperature
    if(OperatingMode == Mode.SwitchWithWBTemperature &&
      airState.WetBulbTemperature < FCStartTemperature)
        return OutletWaterTemperatureFC;

    //Default : Return S.P. in normal mode
    return OutletWaterTemperature;
  }
}
```

Fig. 12 Control schedule of a cooling tower

difficult to manage this class with old simple scheduler programs.

Figure 13 is a sample program that makes a schedule with *Scheduler* class. In this case, the free cooling mode could be activated on the weekdays in winter.

Two kinds of schedules were created from line 3 to 12 in figure 13. A new *Seasons* object that has four seasons was created from line 14 to 16. As described above, *Seasons* object could be treated as *ITermStructure* object, since it implements *ITermStructure* interface. Therefore, *Seasons* object could be used as variable argument for constructor of Scheduler class (line 18 and 19). Similarly, *Days* object was treated as *ITermStructure* object at line 22.

Since the name of *CTSchedule* class was given as a type parameter at a constructer of the *Scheduler* class (line 24 and 25), the *CTSchedule* objects were safely passed as variable argument without type casting at line 27 and 28. At line 31, winter scheduler was related to winter node of parent *Scheduler* recursively.

Figure 14 is a sample program using the scheduler object constructed at figure 13. This program corresponds to the *Client* in the figure 9. The client code can get a *CTSchedule* object by passing *DateTime* object without understanding internal complex structure of *Scheduler* object.

```
 1 public static void MakeSchedulerSample() {
 2   // Schedule which can operate in F.C. mode
 3   CTSchedule fc = new CTSchedule();
 4   fc.OperatingMode =
 5               CTSchedule.Mode.SwitchWithWBTemperature;
 6   fc.OutletWaterTemperature = 32d;
 7   fc.FCStartTemperature = 13d;
 8   fc.OutletWaterTemperatureFC = 16d;
 9   // Schedule which can't operate in F.C. mode
10   CTSchedule noFc = new CTSchedule();
11   fc.OperatingMode = CTSchedule.Mode.NoFreeCooling;
12   fc.OutletWaterTemperature = 32d;
13
14   //Instantiate seasons class (four seasons)
15   ITermStructure terms =
16     new Seasons(Seasons.PredefinedSeasons.FourSeasons);
17   //Create scheduler instance (parent node)
18   Scheduler<CTSchedule> ctScheduler =
19     new  Scheduler<CTSchedule>(terms);
20   //Instatiate days class (week day and week end)
21   terms = new
22        Days(Days.PredefinedDays.WeekDayAndWeekEnd);
23   //Create scheduler instance for winter node
24   Scheduler<CTSchedule> winterSC =
25                    new Scheduler<CTSchedule>(terms);
26   //Operate in F.C. mode on weekdays in winter
27   winterSC.SetSchedule("Weekends", noFc);
28   winterSC.SetSchedule("Weekdays", fc);
29
30   //Composite term (Winter + Weekdays) : Free cooling
31   ctScheduler.SetScheduler("Winter", winterSC);
32   //Others : No free cooling
33   ctScheduler.SetSchedule("Spring", noFc);
34   ctScheduler.SetSchedule("Summer", noFc);
35   ctScheduler.SetSchedule("Autumn", noFc);
36 }
```

Fig. 13 sample program that makes a schedule with Scheduler class

```
public static void GetScheduleSample (
                              Scheduler<CTSchedule> scheduler) {
  DateTime dateTime = new DateTime(1999, 12, 20);
  CTSchedule ctSchedule;
  scheduler.GetSchedule(dateTime, out ctSchedule);
}
```

Fig. 14 sample program using the scheduler object

CONCLUSION

The aim of this study is to accelerate developments of building simulation programs by using Object-Oriented programming. A reusable generalised scheduler interfaces and classes for defining schedules in simulation programs were developed. *ITermStructure* the "interface" for a term structure was developed to make complex term structure general. By using a "Composite-Pattern", all the concrete term classes that implements *ITermStructure* could be integrated into a complex tree structure. By using the "Type parameter", specifications of a schedule class should not be specified until a scheduler classes is declared and instantiated by a client code. Since the program codes developed in this research and a code made by a client were separated clearly, developed scheduler classes were generally applicable for long-term simulation programs. A concrete example of program, which uses the developed scheduler classes to control cooling tower's operating schedules, was given. It demonstrates that the developed scheduler classes worked fine without any modifications by clients.

ACKNOWLEDGEMENTS

This research was partially supported by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Scientific Research (A), No.19206063, 2007.

REFERENCES

J.Page, D.Robinson, N.Morel, J.-L. Scartezzini, 2008, A generalised stochastic model for the simulation of occupant presence, Energy and buildings 40, pp.83-98

Jun Tanimoto, Aya Hagishima, Hiroki Sagara, 2008, Validation of probabilistic methodology for accurate prediction of maximum energy requirements, Energy and buildings 40, pp.316-322

Akito Monden, Daikai Nakae, Toshihiro Kamiya, 2002, Software quality analysis by code clones in industrial legacy software, Proceedings of the 8th Symposium on Software Metrics

Doi Michio, Aman Hirohisa, Yamada Hiroyuki, 2005, A Study of Relationship between Class Reuse Level and Maintainability, Technical report of IEICE. KBSE, (in Japanese)

R. Pressman, 1982, Software Engineering: A Practitioner’s approach, Third edition, McGraw-Hill

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, 1994, Design Patterns Elements of Reusable Object-Oriented Software, Addison-wesley