Time Manager Framework

Antti Viljamaa

Helsinki 2.6.1997

UNIVERSITY OF HELSINKI
Department of Computer Science

1. Introduction

This paper is a product of the FRED project (FRamework EDitor for Java) ongoing at the universities of Helsinki and Tampere. The main goal of the project is to enhance the construction of reusable object-oriented software by improving the tools and technology supporting *frameworks* [Deu89] and *design patterns* [GHJ95, Pre95]. On a general level the project also aims at increasing the overall knowledge of frameworks and design patterns in the industry.

A framework is a set of objects — an application core — that captures the special expertise in some application domain to a reusable form. A whole working application (or a significant portion of it) can be specialized from this skeleton by adding new features to it. The development and use of object-oriented frameworks rely on inheritance, dynamic binding, object composition, and delegation. These basic object oriented features are thus too fine grained to offer significant help for the design process of frameworks. The use of an object oriented language does not guarantee software flexibility in itself. Further design expertise gained from long term experience is usually needed. In this paper a small framework (Time Manager Framework) is presented in order to facilitate how design patterns (which capture "a wealth of experience about the design of object oriented software" [GHJ95]) can be used to make frameworks easily extendible[1].

2. The structure of Time Manager Framework

Time Manager is a framework for applications that resolve time table management problems. One might use this framework for developing applications such as *Headmaster's Solution* and *Appointment Calendar*. Appointment Calendar would allow a project manager to collect calendar information from project members (in distributed machines) and to decide meeting times for the project. Headmaster's Solution would be a more general version of the same application domain. It could enable a headmaster to fit the calendars of several different instances, for example, teachers, classes, and students together to form a proper schedule for the whole school.

A general level OMT class diagram of the framework is shown in figure 1. Abstract methods are written in italics. *Singleton* classes [GHJ95] (classes with only one instance) are marked with an asterisk. The heart of the framework lies within the light grey area. The classes in the darker area are specializations for the application described later.

The class `ResourceProxy` represents a resource that can be reserved for some occasion. It might be, for example, a project worker, a teacher, a set of similar class rooms, a rental car, a group of janitors, etc. The one thing common with resources of different kinds is that their *time* can be allocated for some

[1] It has to be said that the use of design patterns does not eliminate the iterative nature of framework development, but it surely helps in skipping over a few iterations.

occasion. `ResourceProxy` has two immediate subclasses: `MultipleResourceProxy` and `SingleResourceProxy`. The difference between these two is that a `MultipleResourceProxy` can posses several reservation books[2] that describe the time reservations[3] made for that resource, whereas `SingleResourceProxy` has exactly one reservation book. `MultipleResourceProxy` represents a resource that can be in several different places at the same time, i.e. it consists of a group of objects, and it does not matter which particular object is assigned for the job.

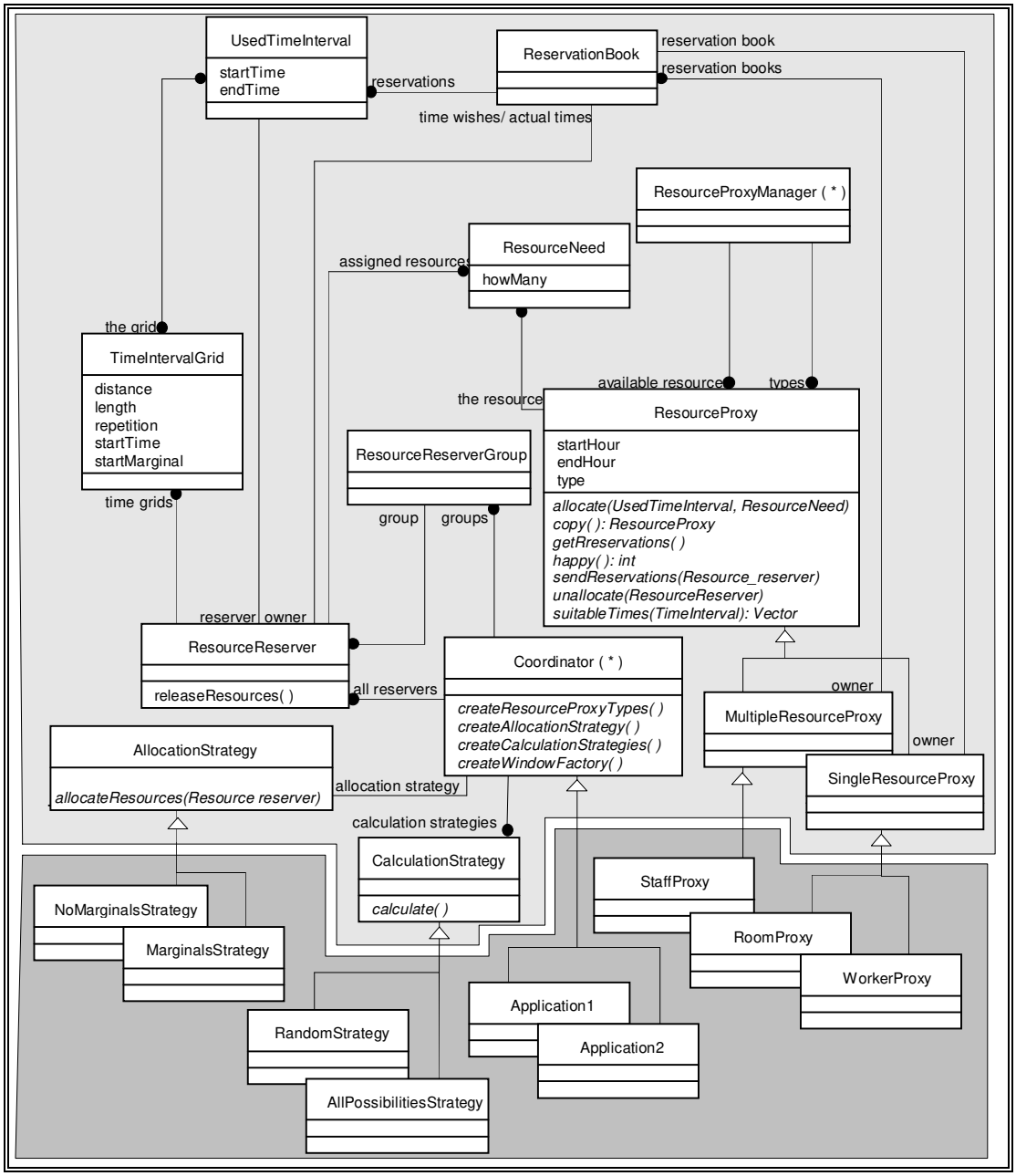

Figure 1: General level OMT class diagram of Time Manager Framework

[2] Instances of the class `ReservationBook`

[3] Instances of the class `UsedTimeInterval`

ResourceProxy includes seven abstract methods that may be implemented in a specific application derived from the framework. The methods allocate and unallocate are used to allocate and release time of the resource. These methods are implemented in the subclasses SingleResourceProxy and MultipleResourceProxy. SingleResourceProxy makes allocations for its one and only reservation book, whereas MultipleResourceProxy tries to allocate time from as many reservation books as defined in the parameter ResourceNeed. Method happy calculates a value representing how pleased the particular ResourceProxy is with the current time table arrangement. For example, a student might be pleased, if the classes are tightly together and there are no holes in the calendar. On the other hand, a rental car is pleased, if its calendar is fully booked. The value returned by the happy method might be used in the algorithm[4] that tries to organize calendars. The method getReservations creates the reservation book(s) for the resources and sets their initial status, i.e. it marks the dates that are already occupied (possibly for personal appointments). The method sendReservations delivers the fixed calendar to the resource. These two methods represent the communication with the actual resource. Somehow the current calendar of the resource must be inputted to the system and the fixed calendar must be sent back to the resource. This can be accomplished in the application derived from the framework, for example, by manually inputting the calendars and by printing the resulted time table, or by using possibly existing database. Another solution would be using the Internet for gathering the calendars of the resources.

The attributes startHour and endHour of the class ResourceProxy represent the "working hours" of a specific type of resource. The attribute type is a string name for the type (class). All these attributes must be instantiated properly in the subclasses.

The class ResourceReserver represents a series of events for which some resources must be allocated. It can be, for example, a series of project meetings or lectures concerning a specific subject of a certain school class. ResourceReserver expresses its resource requirements with instances of the class ResourceNeed. The same resources are allocated for all the events of the series. ResourceReserver belongs to a group of events (ResourceReserverGroup). Different reservers in a group may have different resource requirements.

ResourceReserver has its own reservation book that represents the current time table setting planned for the event series. The times of the events must lie within the constrains represented by instances of the class TimeIntervalGrid. The attribute distance represents the distance between two meetings, length is the length of one meeting, repetition is the number of events, startTime is the full date of the first meeting, and startMarginal defines how much a starting time of an interval can vary.

AllocationStrategy is used to allocate assigned resources for a resource reserver. The method allocateResources must be defined in a subclass. Time allocations of a particular reserver can be released using the method releaseResources.

---

[4] Defined in a subclass of AllocationStrategy

`ResourceProxyManager` takes care of the available resources and handles the creation of new resources. The class `Coordinator` represents the instance that fits different calendars together. It knows the available reservers and reserver groups (objects of the classes `ResourceReserver` and `ResourceReserverGroup`) for which the resources must be allocated and the allocation strategy (an instance of `AllocationStrategy`) used to make the allocations. `Coordinator` also possesses a number of high level calculation strategies (instances of `CalculationStrategy`) used when allocating and releasing resources. `CalculationStrategy` contains an abstract method `calculate` which must defined in the subclasses for proper action.

Figure 2 shows the structure of the user interface. The actual screen elements (windows) are hidden from the core framework behind abstract interfaces. That way the user interface can be altered without modifications to the actual framework.

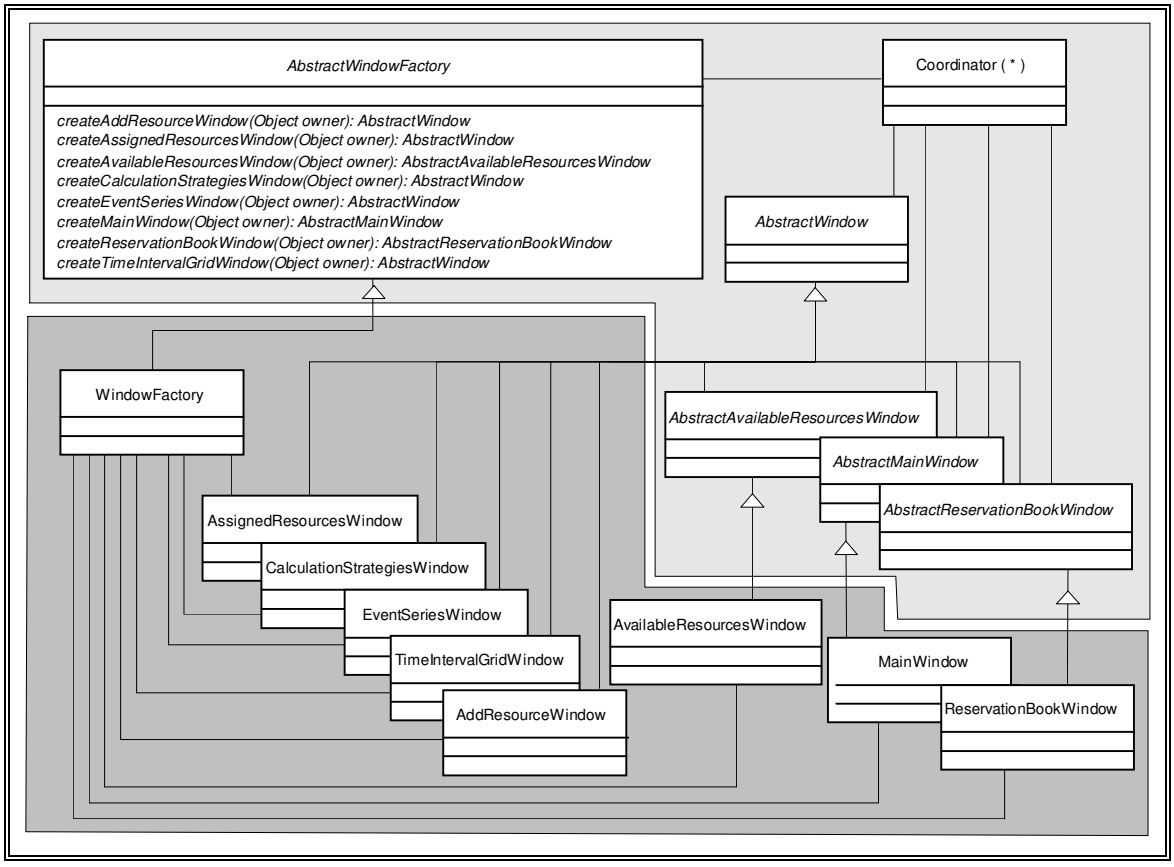

Figure 2: OMT class diagram of the user interface

The design pattern utilized here is called *Abstract factory* [GHJ95]. The class `AbstractWindowFactory` defines abstract methods for creating different user interface windows. A concrete factory (`WindowFactory`) implements these methods so that they return instances of the concrete windows. The core of the framework (represented by class `Coordinator` in figure 2) handles these

concrete windows only through their abstract super classes: it asks `AbstractWindowFactory` to create an instance of a specific window type and receives the appropriate object as a return value.

3. Deriving an application from the framework

Several design patterns have also been utilized when making the actual core framework extendible at the desired *hot spots* [Pre95]. In this framework the hot spots concern the behavior of different types of resource proxies (and their creation), the calculation strategies, the allocation strategy, and the creation of the user interface windows. When hot spots are searched from an application one has to remember not to add flexibility to the framework just for flexibility's sake: flexible solutions also add complexity to the design of a framework and thereby the implementation costs grow very rapidly. Also, we must always carefully consider, how flexible a hot spot should be. Do we have to change the behavior of the program at run time? Should the end-user be able to make the adjustments? Is it enough that the application programmer fixes the possible alternatives?

At the lowest level there are two relevantly different possibilities for making a dynamic decision on which particular method is invocated on a method call [Pre95]. The first possibility[5] is to combine both the *template method* and the *hook method* in the same class[6]. When doing so, adaptations can only be done by overriding the hook method in a subclass. This means that different adaptations require an application restart[7]. The second more flexible solution[8] is to separate the hook method in its own class (= *hook class*). In this way the behavior of a template method can be modified by composition at run time (by plugging in specific instances of defined subclasses of the hook class).

In Time Manager Framework the allocation strategy may need modifications in different applications derived from the framework. Therefore a separation pattern (namely the *Strategy pattern* [GHJ95] from the catalog) can be applied. The hook method `allocateResources` is defined differently in the two existing subclasses: `NoMarginalsStrategy` and `MarginalsStrategy`. `NoMarginalsStrategy` just tries to allocate resources according to the exact time wishes presented by the user. If there are any contradictions, the strategy just prints out an error message telling which reserver already has the particular resource allocated at the given time. `MarginalsStrategy` defines a more sophisticated algorithm, which first collects all the free times of the resources at a given time interval, and then determines the possible common free times and allocates the time slice that is nearest to the original request of the user. If no common free time are available, an error message is displayed.

---

[5] Pree calls this the *unification pattern* [Pre95]. Many of the patterns (f.g. *Template method* and *Factory method*) represented in the pattern catalog [GHJ95] rely on this approach.

[6] A template method defines a complex default behavior which is adjustable by hook methods that the template method calls.

[7] This is true at least in strongly typed languages such as Java and C++.

[8] Pree calls this the *separation pattern* [Pre95]. For example *Abstract factory*, *Builder*, *Command*, *Interpreter*, *Observer*, *Prototype*, *State*, and *Strategy* patterns in the pattern catalog [GHJ95] are based on this solution.

The Strategy pattern is utilized similarly in the implementation of the general calculation strategy. The class `RandomStrategy` redefines the method `calculate` so that it calls the `allocateResources` method for each resource in a random order[9]. `AllPossibilitiesStrategy` tries to make allocations for the reservers in all possible sequence orders. If all allocations succeed in some sequence, the operation is terminated.

In the class `ResourceProxy` a unification pattern (*Template method* [GHJ95]) is used when redefining the behavior of `getReservations`, `happy`, and `sendReservations` in the subclasses. A more complex pattern (*Prototype* [GHJ95]) is needed for creating instances of the subclasses of `ResourceProxy` chosen at run time. The pattern works in the following manner. The super class `ResourceProxy` defines an abstract method `copy` which returns a new instance of `ResourceProxy`. The method is redefined in the subclasses to return an instance of the specific subclass. A new instance is created by cloning an existing object of a desired class. The initial instance of every subclass is created when instantiating the whole application (in a subclass of `Coordinator`, as explained later). When the first object of each subclass of `ResourceProxy` is created, the instance registers itself to `ResourceProxyManager`, which holds the clonable prototypes. `ResourceProxyManager` associates a string name with each subclass type, so that the end user can choose which class to instantiate at run time.

The application programmer can use the available subclasses (`NoMarginalsStrategy`, `MarginalsStrategy`, `RandomStrategy`, `AllPossibilitiesStrategy`, `StaffProxy`, `RoomProxy`, and `WorkerProxy`) to compose a working application, or she can naturally define her own subclasses. Either way the instantiation of chosen subclasses is done in a subclass of `Coordinator` (named as, for example, `Application1` as in the sample code provided in figure 3).

```
package Application;

import TimeManager.*;
import UserInterface.*;

public class
Application extends Coordinator {

  public
  AllocationStrategy
  createAllocationStrategy() {
    return new MarginalsStrategy();
  }

  public void
  createCalculationStrategies() {
    new RandomStrategy();
    new AllPossibilitiesStrategy();
  }

public void
  createResourceProxyTypes() {
    new WorkerProxy();
    new RoomProxy();
    new StaffProxy();
  }

  public
  AbstractWindowFactory
    createWindowFactory() {
      return new WindowFactory();
  }

  public
  static void
    main(String[] argv) {
      instance_ = new Application();
      instance_.initialStuff();
  }
}
```

Figure 3: The initialization code for a running application

[9] Random calculation makes sense in that the result may depend on the order in which the allocations are made for each reserver.

`Coordinator` is implemented using a unification pattern (*Factory method* [GHJ95]), so that the creation of objects is defined in the subclasses. For example the first instances of the available resources are created in the method `createResourceProxyTypes`, the allocation strategy is created in the `createAllocationStrategy` method, available calculation strategies are instantiated in the method `createCalculationStrategies` and the concrete window factory is created in the method `createWindowFactory`. Like different resource proxy types, also calculation strategies register themselves automatically to the core framework during their creation. This is implemented in the constructor of their super class.

Also other design patterns have been utilized to make the framework more flexible and maintainable. These include for example *Singleton*[10], *Proxy*[11], and *Observer*[12].

Time Manager framework can be considered as a black-box framework, because the central classes of the framework can be interpreted in different flexible ways in different applications, so in a simple case an application can be constructed just by creating proper instances of the classes already included in the framework. On the other hand, Time Manager framework is a white-box framework, because the classes concerning for example the communication with the resources and different strategies for organizing the time tables can be declared in the portion provided by a specific application.

4. A running application

Figure 4 shows an example screen shot of a running application derived from the framework. The picture corresponds to the initialization code represented in figure 3. In figure 4 the user has specified two resource reserver groups (*Personal* and *FRED-project*). Group FRED-project has three subreservers (*Management meeting*, *Workers weekly*, and *Combined*). The asterisks before the reserver names denote successful resource allocations[13]. The window at the background represents the time allocations made for the resource *Tauno Kayhko*. The window in front of that includes all times specified for the reserver FRED-project. The frontmost window shows all the available calculation strategies.

The uppermost window in the figure (main window) allows the user to define new event series for selected reservers (*Event series*), show all existing events (*All events*), assign new resources for the reserver (*Assign resources*), or manually allocate and release the assigned resources (*Allocate resources*, *Release*

[10] Singleton pattern advises how to construct a class with only one possible instance and a global access to it [GHJ95]. Singleton classes in Time Manager Framework are `Coordinator` and `ResourceProxyManager`.

[11] Proxy pattern advises, for instance, that the creation of large objects should be delayed until they are truly needed. At the mean time a proxy can represent them. In Time Manager Framework the creation of a user interface window is delayed until it is actually first shown on the screen. Also resources (possible existing in other computers) are represented as proxies (`ResourceProxy`) in the framework.

[12] The Observer pattern is used to define one to many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. In Time Manager Framework user interface windows are Observers for their proxies. Proxies contain the actual data to be represented in the windows.

[13] An exclamation point would denote a failed allocation.

resources). From the *File* menu one can save the current state of the application to a file or load a previously saved file. *Edit* menu allows the user to add or remove reserver groups and reservers, *Show* menu gives access to available resources and calculation strategies (resources can be added or removed). *Calendar* menu is for sending personal appointment times to an other machine and similarly to receive the reservations from someone else. Calendar menu commands are left abstract and they are not currently implemented in any of the subclasses of `ResourceProxy`.

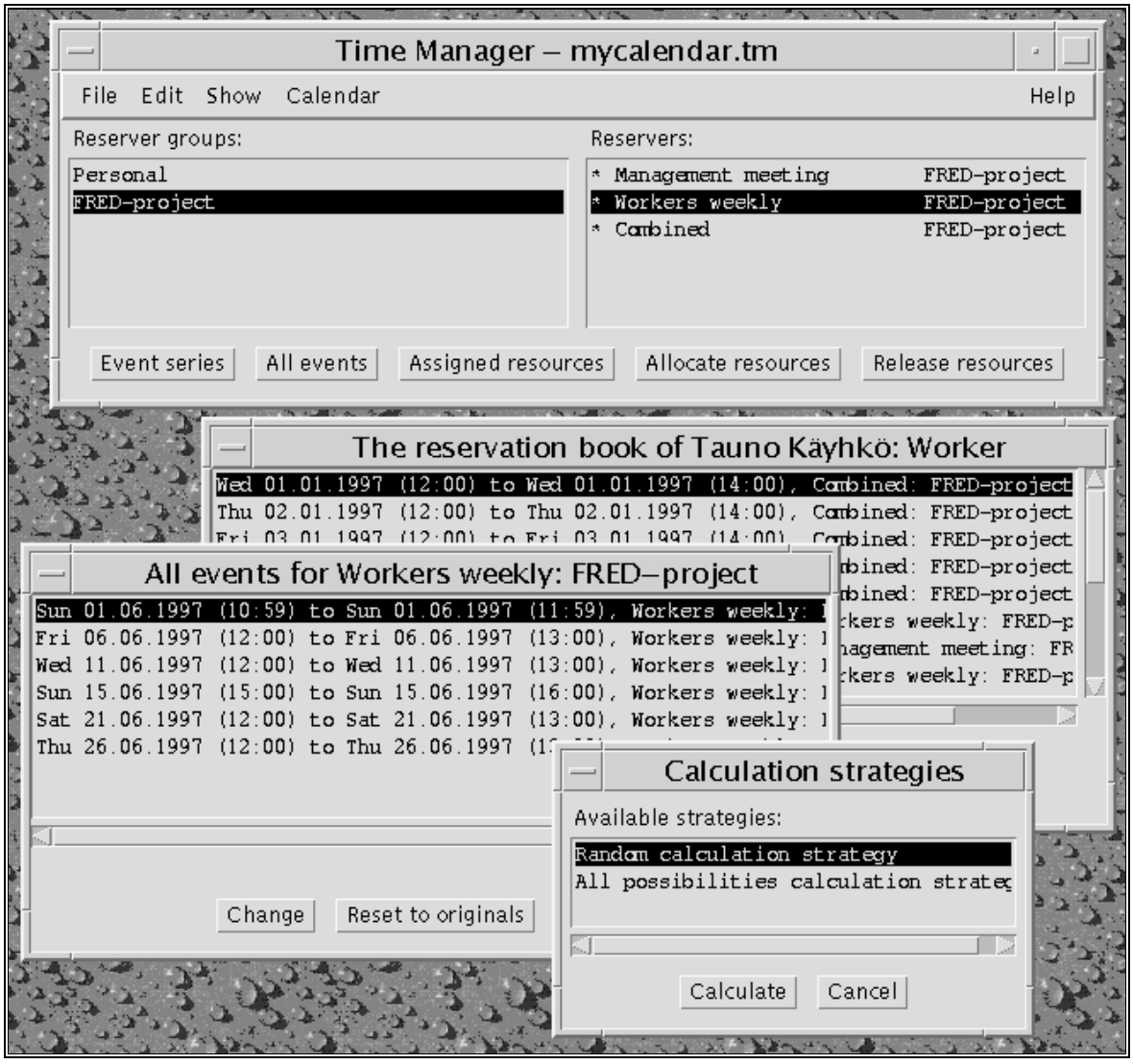

Figure 4: A running application derived from the framework

References

Deu89 Deutsch L., Design Reuse and Frameworks in the Smalltalk-80 System. In: *Software Reusability Vol. II* (Biggerstaff T., and Perlis A., eds.), ACM Press 1989, 57-71.

GHJ95 Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

Pre95 Pree W., *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1995.