Recurring Events

Martin Fowler
100031.3311@compuserve.com

I had a happy time when I lived in the South End of Boston. I had a nice apartment in an attractive brownstone building. It's not the poshest part of town, but it was lively and a short walk from most things I needed. I did have a few irritations, however, and one of these was street cleaning. Now I like to have clean streets, but I also had to park my car on the street (off-street parking is incredibly expensive). If I forgot to move my car, I got a ticket, and I often forgot to move my car.

Street cleaning outside my old house occurs on the first and third Monday of the month between April and October, except for state holidays. As such its a recurring event, and recurring events have been a recurring event in my modeling career. An electricity utility I worked with sends out their bills on a schedule based on recurring events, a hospital books its out-patient clinics on a schedule based on recurring events, a payroll is run on a schedule based on recurring events, indeed employees' pay is based on rules that are often based on recurring events.

This pattern language describes how we can deal with these recurring events in a computer system, so that the computer can figure out when various events occur. We begin by looking at where the responsibility should lie for working them out. *Schedule (1)* suggests that we define a specific class to handle the understanding of recurring events, so any objects that needs to deal with them (whether a doctor or a street) can do say by being given a schedule. It can be tricky to see what this schedule object looks like, however, especially if you tend to think of objects in terms of their properties. *Schedule's Interface (2)* allows you to think about what you want from the schedule rather than how you set up a schedule.

With the interface in hand, we can now begin to think about schedule's properties. A schedule needs to work out which events (there may be several) occur on which days. *Schedule Element (3)* does this by giving a schedule a schedule element for each event, with the 'when' part delegated to a temporal expression. A temporal expression has an individual instance method [2] to work out whether a day lies within the temporal expression or not. At this point we separate the (simple) event matching from the (tricky) time matching.

We could come up with some language for defining temporal expressions, or some powerful class that can be used to handle the rather wide range of temporal expressions that we need to deal with. However I'm not inclined to develop a complex general solution if I can think of a simple solution that solves *my* problem. Such a simpler solution is to think of some simple temporal expressions, and define subclasses of temporal expression for them. *Day Every Month (4)* handles such expressions as second monday of the month . *Range Every Year (5)* deals with such things as between 12 April and 4 November each year. I can then combine these temporal expressions with *Set Expression (6)* to develop more complex cases, such as my old street cleaning. Using set expressions in this way is a simple application of interpretor [3].

1 Schedule

My friend Mark is a physician in a London hospital. On the first and third monday of the month he has a gastro clinic. On the second wednesday of the month he has a liver clinic. (Actually I don t know what his real schedule is and I m making this up, but I m sure you ll forgive me.) His boss may have a liver clinic on the second and fourth Tuesdays of a month, and golf on every monday (hospital consultants seem to do a lot of research on golf courses, I guess the swing is good for their technique).

One way of representing this might be to consider that Mark has a set of dates for each event (Figure 1). This supports our needs, since we can now easily tell the dates of Mark s clinics, but it comes with its own problems.

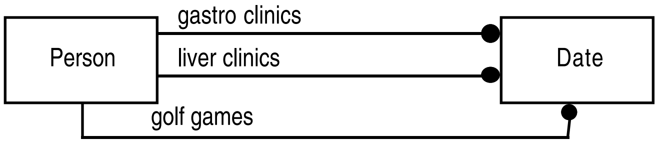


Figure 1. OMT[4] object model for a person with an association for each event.

The first problem is that when we have an association for each event that Mark has, we have to modify the model for each change we make. Should our doctors get a new kind of clinic, or take up gliding, we have to add an association, which implies changing the interface of the person class.

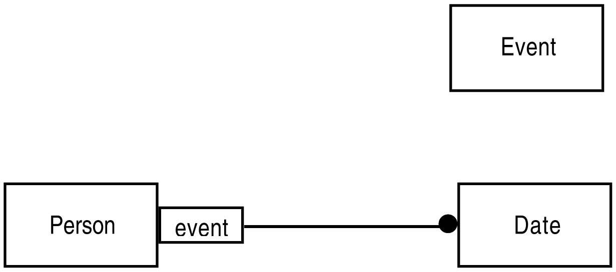


Figure 2. Person with a qualified association to date

Figure 2 deals with this problem by using a qualified association. It defines a new type, event, and says that each person has a set of dates for each instance of event (qualified association are talked about in more detail in [2] as keyed mappings, they correspond to Smalltalk dictionaries or C++ STL maps). Now whenever we get some new clinic, all we have to do is create a new instance of event, which deals well with that problem. (For the purposes of this paper, event is just a string representing some activity, like a clinic or golf game, but in practice it can get rather more involved.)

Another problem is to ask how we would set up the dates? Do we actually want to imply that we have to assert the individual dates for the person. We would prefer to just say every second monday . Bear with me on that one, I ll come to it later.

Figure 2 is certainly heading in the right direction, but I m not comfortable with the responsibility on person. I can imagine many questions you might want to ask regarding the dates, and loading all that stuff onto person is awkward, because person will usually has enough to do in any system I come across. Also you will find other objects that might have similar behavior, such as my street.

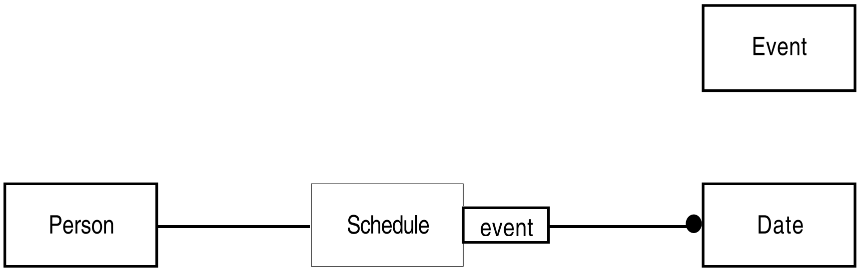


Figure 3. Using schedule as a separate object.

So I m inclined towards Figure 3 which puts all the responsibility of tracking dates and events on a separate type: schedule. Now if we want some type to have this behavior we just give them a schedule.

2 Schedule s Interface

What kind of questions are we going to ask the schedule? Schedule is one of those objects that can really trip up people like me who have come from a data modeling / database background. This is because that training makes us want to look at schedule in terms of its properties. Providing we are using taking a conceptual perspective, and not getting hung up on what is stored and what is calculated; this is not too much of a problem, at least for information systems. I find that schedule is one of the exceptions, for whenever I have worked with it I get frazzled.

Thinking of an object through its properties is a very natural way to think of something. It allows us both to query and change the object easily. When frazzling occurs, however, then that is a sign to try another tack. At this point I look at how I might use a schedule once I have one. I forget about its internal structure, I also forget about how I set one up. Both of those are secondary to using a completed schedule, so I just assume its set up by magic and ask myself what I want it to do.

I doubt if I would really want Mark s schedule to tell me all the days he is due for a gastro clinic. I might want to know which days he was booked this month, but not from the epoch to the end of time. So one question would be Occurrences (Event, DateRange) which would return of set of dates. Another would be to find out when his next gastro clinic is scheduled, this might be from today, or from another date: nextOccurrence (Event, Date). A third would be to determine whether an event would occur on a given date: isOccurring(Event, Date). Naturally you would examine your use cases to come up with some more, but we don t want the full list, merely the core items (Listing 1). As it is these are not minimal, in that two could be defined on top of one of them. Getting a few of them gives me a better feeling of how schedule will be used. Now I have a sense of where to go next because I know what I want to aim at next.

```
class Schedule {
    public boolean isOccurring(String eventArg, Date aDate)
    public Vector dates (String eventArg, DateRange during)
    public Date nextOccurence (String eventArg, Date aDate)
};
```

Listing 1. Java[1] interface for schedule

3 Schedule Element

With some picture of an interface we can now begin to think about how a schedule is created. The main point of a schedule is that it tracks a correspondence between events and dates, and does so in such a way that the dates can be specified by some expression. This leads me to a schedule containing elements, each of which links an event to an expression that determines the appropriate dates. Whenever an expression rears its head, properties cause more trouble than they are worth, so again I think of an interface. This expression should have some way of telling whether a particular date is true or not. Thus each instance of this temporal expression will have a method that takes a date and returns a boolean (Figure 4). This is an example of the Individual Instance Method pattern [2], each schedule element would have its own method for determine whether or not a date fits it. Conceptually we can think of the method as a block of code, but one that is different for each instance. A regular instance method (or member function) executes against an instance but is the same for all instances of the class. In this case the method executes against an instance and is different for each instance. I will look at implementing this a little later, again sorting out the interface is important.

> **Example:** Mark has a gastro clinic on the first and third monday of the month, and a liver clinic on the second wednesday. This would be represented by a schedule with two schedule elements. One schedule element would have an event of gastro clinic and a temporal expression that would handle the first and third monday of the month. The other schedule element

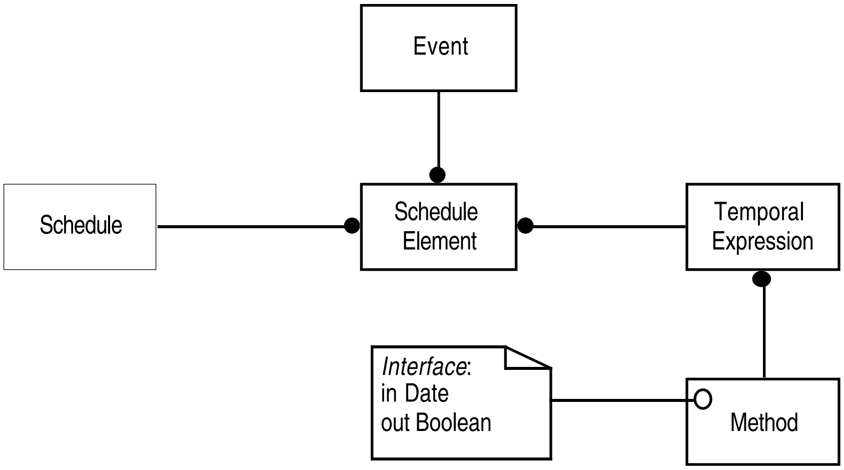
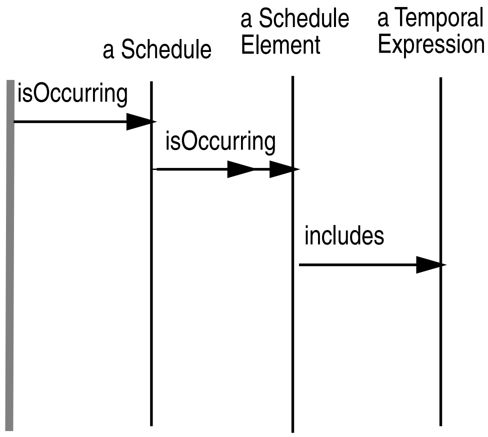


Figure 4. Schedule Element

would have an event of liver clinic and a temporal expression that would handle the second wednesday of the month.

Here the dynamic behavior is getting interesting. The core behavior is that of responding to isOccurring.The schedule delegates the message to its elements. Each element checks the event for a match and asks the temporal expression if the date matches. The temporal expression thus needs to support a boolean operation includes (Date). If the event matches and the temporal expression reports true then the element replies true to the schedule. If any element is true then the schedule is true, otherwise it is false. (Figure 5, and Listing 2)



A schedule is asked to check an event on a date.

It asks each schedule element to check the event and date.

The schedule element sees if the event is the same and gets the temporal expression to test the date.

If any schedule element replies true then so does the schedule, otherwise it replies false

Figure 5. Interaction diagram to show how a schedule finds out if an event occurs on a date.

The patterns have brought us to a point where the problem of considering the event is separated from that of forming the temporal expression. All we need to do know is figure out how to form the temporal expression, and all is dandy.

```
class Schedule {
    public boolean isOccurring(String eventArg, Date aDate) {
        ScheduleElement eachSE;
        Enumeration e = elements.elements();
        while (e.hasMoreElements()) {
            eachSE = (ScheduleElement)e.nextElement();
            if (eachSE.isOccurring(eventArg, aDate))
                return true;
        }
        return false;
    };
class ScheduleElement {
    public boolean isOccuring(String eventArg, Date aDate) {
        if (event == eventArg)
            return temporalExpression.includes(aDate);
        else
            return false;
    };
```

Listing 2. Java method to determine if an event occurs on a date

4 Day Every Month

So far we have a temporal expression which can say true or false for any given day. Now the question is how do we create such thing? The conceptually simplest idea is to have a block of code for each object, conceptually simple but rather awkward to implement. We could develop some interpreter that would be able to parse and process a range of expressions that we might want to deal with. This would be quite flexible, but also pretty hard. We could figure out some way to parameterize the object so that all possible expressions could be formed by some combination of properties. This may be possible, it certainly would be tricky.

Another approach is to look at some of kinds of expression that this system has to deal with, and see if we can support them with a few simple classes. The classes should be as parameterized as possible, but each one should handle a particular kind of expression. Providing they all respond to includes, this will work. We may not be able to cover everything that we can conceive of, at least not without creating a new class, but we may well be able to cover pretty much everything with a few classes.

The first such animal is to cope with phrases like first monday of the month. In a phrase such as this we have two variables: the day of the week, and which one we want in the month. So our day in the month temporal expression has these two properties (Figure 6). Internally includes uses these to match the date (Listing 3).

> **Example:** Mark has a gastro clinic on the second monday of the month. This would be represented using a day in month temporal expression with a day of the week of monday and a count of 2. Using Listing 3 this would be DayInMonthTE (1, 2).
>
> **Example:** Mark also has a liver clinic on the last friday of the month. This would be represented using a day in month tem-

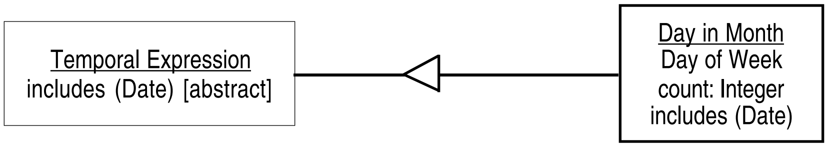


Figure 6. Day in month temporal expression

```
abstract class TemporalExpression {
    public abstract boolean includes (Date theDate);
}

class DayInMonthTE extends TemporalExpression{
    private int count;
    private int dayIndex;
    public DayInMonthTE (int dayIndex, int count) {
        this.dayIndex = dayIndex;
        this.count = count;
    };
    public boolean includes (Date aDate) {
        return dayMatches (aDate) && weekMatches(aDate);
    };
    private boolean dayMatches (Date aDate) {
        return aDate.getDay() == dayIndex;
    };
    private boolean weekMatches (Date aDate) {
        if (count > 0)
            return weekFromStartMatches(aDate);
        else
            return weekFromEndMatches(aDate);
    };
    private boolean weekFromStartMatches (Date aDate) {
        return this.weekInMonth(aDate.getDate()) == count;
    };
    private boolean weekFromEndMatches (Date aDate) {
        int daysFromMonthEnd = daysLeftInMonth(aDate) + 1;
        return weekInMonth(daysFromMonthEnd) == Math.abs(count);
    };
    private int weekInMonth (int dayNumber) {
        return ((dayNumber - 1) / 7) + 1;
    };
```

Listing 3. Selected Java code for a day in month temporal expression.

Java s date class represents day of the week using an integer range 0—6 for sunday—saturday. I have used the same convention.

poral expression with a day of the week of friday and a count of -1.

5 Range Every Year

Some events can occur in a particular range in a year. In a British government establishment they had set dates for turning the heating on and off (they didn't respond to anything as logical as temperature). To handle this we can use another subtype of temporal expression, this one can set up with start and end points, using a month and a day (Figure 7). We can create one of these expressions several ways, depending on whether we need date precision or not (Listing 4). A common need is to indicate just a single month, as we shall see later. The includes method now just looks at the date and tests whether it fits within that range (Listing 5).

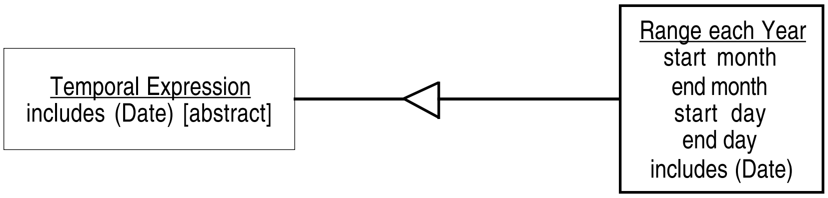


Figure 7. Range each year temporal expression.

```
public RangeEachYearTE (int startMonth, int endMonth,
                        int startDay, int endDay) {
    this.startMonth = startMonth;
    this.endMonth = endMonth;
    this.startDay = startDay;
    this.endDay = endDay;
};
public RangeEachYearTE (int startMonth, int endMonth) {
    this.startMonth = startMonth;
    this.endMonth = endMonth;
    this.startDay = 0;
    this.endDay = 0;
};
public RangeEachYearTE (int month) {
    this.startMonth = month;
    this.endMonth = month;
    this.startDay = 0;
    this.endDay = 0;
};
```

Listing 4. Creating a range each year temporal expression
If no date is specified it is set to zero.

Example: The heating is turned off on the 14 April and turned on the 12th October. This could be represented as a range each year temporal expression with a start month of April, start date of 14, end month of October, and end date of 12. Using RangeEachYearTE it would be set up with RangeEachYearTE (3, 9, 14, 12)[1]

```
public boolean includes (Date aDate) {
    return monthsInclude (aDate) ||
      startMonthIncludes (aDate) ||
      endMonthIncludes (aDate)
};
private boolean monthsInclude (Date aDate) {
    int month = aDate.getMonth();
    return (month > startMonth && month < endMonth);
}
private boolean startMonthIncludes (Date aDate) {
    if (aDate.getMonth() != startMonth) return false;
    if (startDay == 0) return true;
    return (aDate.getDate() >= startDay);
}
private boolean endMonthIncludes (Date aDate) {
    if (aDate.getMonth() != endMonth) return false;
    if (endDay == 0) return true;
    return (aDate.getDate() <= endDay);
}
```

Listing 5. The includes method for RangeEachYearTE

6 Set Expression

The temporal expressions above provide some ability to represent the kinds of problem we deal with, but we can greatly enhance their abilities by combining them in set expressions (Figure 8 and Listing 6). Set expressions require three classes: union, intersection, and difference. Each set expression holds a number of components and processes them in the appropriate way. If we make these three classes composites[3], we can put set expressions within set expressions; which allows us to build quite complex expressions. This is a useful technique whenever you want to combine some kind of selection expression (you can also think of these as boolean operations).

> **Example:** The US holiday of memorial day falls on the last monday in May. This can be represented by an intersection temporal expression. Its elements are a day in month with count -1 and day of week of monday, and a range every year with start and end month of may.
>
> **Example:** Street cleaning occurs from April to October on the first and third mondays of the month, excluding state holidays. The representation is rather tricky to describe in words, so take a look at Figure 9, the code is in Listing 8.

Using set expression in this way is a use of the Interpretor pattern [3]. It is interesting to note that I didn t realize this until Ralph Johnson pointed it out to me. In my mind interpretors are for languages, and languages are complicated. This is simple, and yet the interpretor pattern still works very well, so well

---

1. Yes the months are correct. Java s date class represents months with an integer of range 0—11.

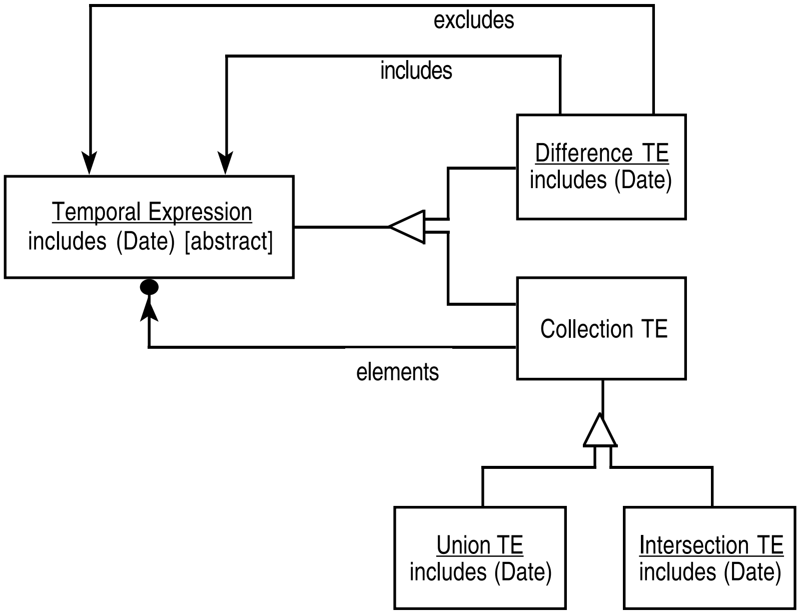


Figure 8. Set expressions

```
class UnionTE
    public boolean includes (Date aDate) {
        TemporalExpression eachTE;
        Enumeration e = elements.elements();
        while (e.hasMoreElements()) {
            eachTE = (TemporalExpression)e.nextElement();
            if (eachTE.includes(aDate))
                return true;
        }
        return false;
    };
class IntersectionTE
    public boolean includes (Date aDate) {
        TemporalExpression eachTE;
        Enumeration e = elements.elements();
        while (e.hasMoreElements()) {
            eachTE = (TemporalExpression)e.nextElement();
            if (!eachTE.includes(aDate))
                return false;
        }
        return true;
    };
class DifferenceTE
    public boolean includes (Date aDate) {
        return included.includes(aDate) && !excluded.includes(aDate);
    };
```

Listing 6. Includes methods for the set expressions

```
        IntersectionTE result = new IntersectionTE();
        result.addElement(new DayInMonthTE(1,-1));
        result.addElement(new RangeEachYearTE (4));
        return result;
```

Listing 7. Code for creating a temporal expression for memorial day.

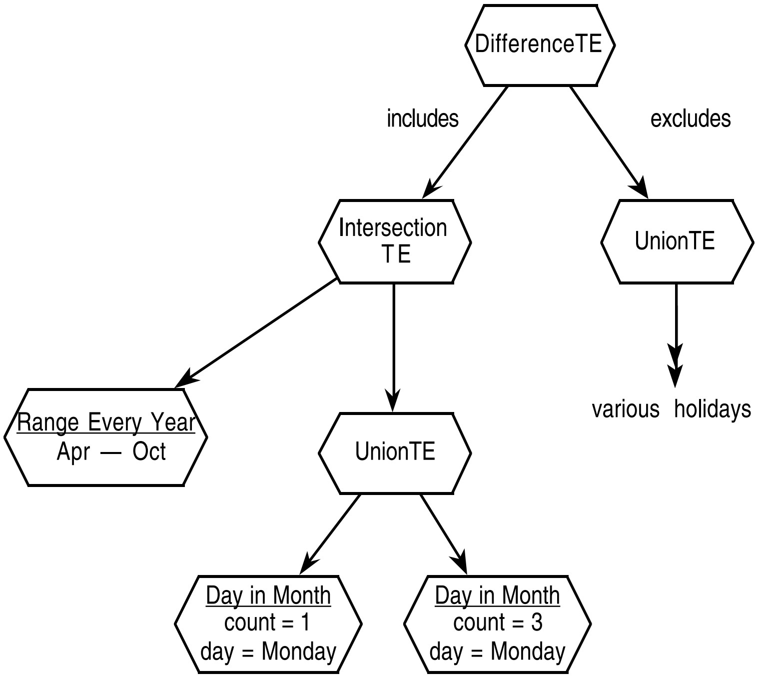


Figure 9. Instance diagram showing objects to represent a street cleaning schedule

```
    public DifferenceTE streetCleaning() {
        UnionTE mon13 = new UnionTE();
        mon13.addElement(new DayInMonthTE(1,1));
        mon13.addElement(new DayInMonthTE(1,3));
        IntersectionTE nonWinterMons = new IntersectionTE();
        nonWinterMons.addElement(mon13);
        nonWinterMons.addElement(new RangeEachYearTE (3,9));
        return new DifferenceTE(nonWinterMons, maHolidays());
    }
```

Listing 8. Java code for the street cleaning schedule

that it is easy to use it without realizing it, which I guess is the sign of a good pattern!

Going Further

Time and the PLoP limits bring to a halt here, however I should not stop without indicating some further patterns that need to be developed.

- When holidays occur they may cancel out the recurring event (as occurs in street cleaning). But a substitute may occur, such as do it the following Monday, or the next Thursday.
- The patterns here concentrate on events, but they can also be used to handle defining days as working days, or further ways to classify days. This may be as simple as every Monday to Friday is a working day.
- Some some events should not occur on the same day. Can we do something about this, or just trust our ability to write good temporal expressions?
- How do we handle a schedule such as four weeks on two weeks off?

References

1 Arnold, K. and Gosling, J. *The Java Programming Language*, Addison-Wesley, Reading, MA, 1996.
2 Fowler, M. *Analysis Patterns: reusable object models*, Addison-Wesley, Reading MA, in press.
3 Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns: elements of reusable object-oriented software* , Addison-Wesley, Reading, MA, 1995.
4 Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.

	Name	Problem	Solution
1	Schedule	Someone has events which occur on certain recurring days.	Create a schedule object for the doctor which can say which days an event occurs on
2	Schedule s Interface	It is hard to see what schedule should look like.	Imagine a schedule object is already created and consider how you would use it. Determine the key operations in its interface.
3	Schedule Element	You need to represent recurring days without enumerating them	Schedule Element with event and temporal expression. Temporal expression has an individual instance method to determine if dates match.
4	Day Every Month	You need to represent statements of the form 2nd Monday of the Month	Use a day every month temporal expression with a day of the week and a count
5	Range Every Year	You need to represent statements of the form 14 March till 12 October	Use a range every year temporal expression with a day and month at the start and end
6	Set Expression	You need to represent combinations of temporal expressions	Define set combinations for union, intersection and difference

Table 1: Table of Patterns