IMPLEMENTING A DIGITAL CALENDAR IN A PORTFOLIO MANAGEMENT CONTEXT USING TEMPORAL EXPRESSIONS

BY

KATHLEEN CHALAS

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Adviser:

Research Associate Professor Ralph E. Johnson
Abstract

Different systems implement digital calendars according to their context-specific needs. In the case of a portfolio management system for the financial industry, a digital calendar needs to keep track of financial events that can be both recurring and nonrecurring as well as provide business analysts with a method of querying calendars about those events. The patterns to define those events need to be flexible enough to create complex and simple recurrences, such as “every second Tuesday of the month” or “every Christmas”. With a system requirement that exposes the construction of patterns that are then given to define specific events - such as to define business days or fiscal periods - the interface also needs to be simple enough to seem familiar to accountants who are used to filling out spreadsheets and programming them with short scripts or equations. Martin Fowler’s Schedule design pattern provides a model for a flexible calendar design that, with some tweaking, can meet all these requirements. Though calendars are common and prevalent, the context and, by extension, its specific use cases heavily inform the design. This thesis provides a look into the process of implementing a calendar in a financial context in its full process, from considering requirements to implementation to benchmarked analysis of performance.
# Table of Contents

List of Tables........................................................................................................................................ iv  
List of Figures.......................................................................................................................................... v  
List of Abbreviations .............................................................................................................................. vi

## Chapter 1  Introduction ....................................................................................................................... 1  
  1.1 Motivation ...................................................................................................................................... 1  
  1.2 Related Work ................................................................................................................................. 3

## Chapter 2  Background .......................................................................................................................... 6  
  2.1 Overview of Design of Surrounding System .................................................................................. 7  
  2.2 Joda-Time ..................................................................................................................................... 9

## Chapter 3  Requirements ..................................................................................................................... 11  
  3.1 Division of Time ............................................................................................................................. 11  
  3.2 Support Singular and Recurring Events ....................................................................................... 11  
      3.2.1 Querying Events .................................................................................................................. 12  
      3.2.2 Flexible Rules for Recurring Events .................................................................................. 13  
  3.3 Interface as Models for User Benefit ............................................................................................ 14

## Chapter 4  Implementation ................................................................................................................ 16  
  4.1 High-Level Overview of Design ................................................................................................. 17  
  4.2 Calendar and Events ..................................................................................................................... 20  
  4.3 Temporal Expressions ................................................................................................................... 25  
  4.4 Fiscal Periods ............................................................................................................................... 33  
  4.5 Design Decisions Concerning End Users .................................................................................... 37

## Chapter 5  Performance ....................................................................................................................... 44  
  5.1 Results and Analysis ...................................................................................................................... 50

## Chapter 6  Conclusions and Future Work ......................................................................................... 55

References .................................................................................................................................................. 57
List of Tables

Table 1: DataSet singular specifier constructors reformatted for end users ........................................ 42
Table 2: Description of benchmarks by types and numbers of IntervalGenerator and CalendarEvent objects and number of dates being searched ................................................................. 49
Table 3: Exact numbers for all ten runs of benchmarks, standard deviation, and garbage collection ............................................................................................................................................. 51
Table 4: IntervalGenerator benchmark run results with cache implemented ........................................... 52
Table 5: Average results of ten runs of nextDatesFor() benchmarks ....................................................... 54
List of Figures

Figure 1: Class Diagram of Calendar in MetaBooks ................................................................. 18
Figure 2: Java code showcasing two different addEvent() methods ............................................ 21
Figure 3: Sequence diagram of eventsFor() called on the Calendar class ................................. 23
Figure 4: Java code for creating a business day pattern in a JUnit test ..................................... 31
Figure 5: Java code for creating a pattern of one-time holiday events using Add ...................... 32
Figure 6: newCalendarEvent() method called by eventsFor() in IntervalGenerator ................. 35
Figure 7: Groovy code for defining an IntervalGenerator ......................................................... 36
Figure 8: Old and new syntax for defining a business day pattern ........................................... 38
Figure 9: Using Groovy extensions to extend foreign classes to avoid imports ......................... 40
Figure 10: Annotation for adding benchmarking to test cases ............................................... 46
Figure 11: Average time in seconds for each calendar to find 15 LocalDates ............................. 50
List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>PMS</td>
<td>Portfolio Management System</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface</td>
</tr>
<tr>
<td>SE</td>
<td>Standard Edition</td>
</tr>
<tr>
<td>FTP</td>
<td>File Transfer Protocol</td>
</tr>
<tr>
<td>GC</td>
<td>Garbage Collection</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>GB</td>
<td>Gigabytes</td>
</tr>
<tr>
<td>GHz</td>
<td>Gigahertz</td>
</tr>
<tr>
<td>PDA</td>
<td>Personal Digital Assistant</td>
</tr>
<tr>
<td>IMAP</td>
<td>Internet Message Access Protocol</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Calendars have been used for thousands of years by people all over the world to keep track of things like holidays, celebrations, religious events, or payment schedules. Builders use calendars to estimate and schedule completion times for architectural schematics. Marketing experts use calendars to consider the deployments of advertisements for products they represent as well as the products of their rivals. Farmers use calendars to track seasons and plan harvests. The justice system uses calendars to organize the trials and sentencings of people appearing in court. Programmers use calendars to plan sprints in the agile methodology or lay out release schedules for software patches. People from all walks of life use calendars, be they young students marking dates with field trips and test schedules or experienced professionals keeping track of meetings and contacts. But, in this digital age, people do not generally walk around with printed calendars tucked into their pockets. Instead, they use digital calendars, which are typically embedded in a host of applications and specialized for ease of use in particular context-dependent tasks.

1.1 Motivation

The implementation of a functioning and useful calendar, on first approach, can seem trivial simply by the prevalence of calendar systems in the everyday environment. There is the calendar that hangs on the wall, filled with penciled in events, appointments, and notes detailing locations, phone numbers, and other relevant information. Then there is the calendar on a phone’s home screen or a game system’s dashboard, informing users of the date and time and
any appointments scheduled on this particular date. Operating systems typically sport calendars on their task bars, pre-mounted for easy and quick access. In the cloud, there are shared calendars that can support combining the calendars of friends, online communities, and organizations as well as providing alerts before beginnings of events. In an email client, there may be another calendar, specific to an organization or shared amongst a group. In an educational institution, each class has its own calendar with lectures, assignments, and deadlines attached to dates.

The more prevalent the presence of something, the easier it is to underestimate its importance. Without special calendars at different levels of a system – from the one on the device, to the one on the email client, to the one in the cloud – it would be increasingly difficult to organize and categorize events. Particularly since a task-oriented approach to work is more common. People have the tendency to immerse themselves into a single task at a time. By having related tasks, contacts, and appointments on a calendar, other reminders are not present to distract someone interested in completing a particular task. Programming sprints do not appear with reminders to do laundry or attend a wedding; instead, only the events related to completing development and testing appear on a sprint calendar. And just as it is useful to have multiple calendars on a task-by-task basis, it is also useful to have embedded calendars in software applications.

If the only access a user had to a calendar was from the dashboard of their laptop or phone, then there would be a requirement to always check this centralized location and search through all the calendars a user might have to find the desired one. Such a requirement would be disruptive to workflow and overtask a user with the necessity to organize the different calendars in such a way that they would be easy to search through. It may also place sensitive information
in the same space as inconsequential information. For instance, a doctor is tasked with keeping doctor-patient confidentiality. In placing the appointment calendar, including any notes that might be made about the nature of the appointment, in the same location as public calendars, such as a schedule of theatrical events and fundraisers, or even shared calendars between peers or family, the doctor risks making a mistake in setting permissions or mistaking the one calendar for another. Offices typically employ their own software for scheduling, and banking applications generally have a date-awareness and a means of keeping track of dates and payments within the context of a financial software. The convenience, reduced risk of mistakes, and reduced task disruption of context-specific calendars makes them worth the effort to implement in applications and there is a variety of ways to go about implementing calendars depending on the needs of the context.

In the context of a financial application, the calendar component described in this thesis is one modeled from Martin Fowler’s Schedule design pattern. This thesis is organized to first introduce other research that has focused on calendar design in the rest of chapter one. Then the financial background is introduced in chapter two, along with the system in which this implementation of a calendar is a component. Chapter three describes the specific requirements that are necessitated or desired in a financial context for business analyst end users. Chapter four proceeds to detail the implementation and design of the calendar. Chapter five focuses on benchmarking very iterative code and analyzes benefits of refactoring any slower processes. Finally, chapter six concludes the thesis.

1.2 Related Work
Though at the mention of a calendar, the first mental image that arises is a wall calendar hung on a refrigerator and covered in notes, even in physical calendars there is some ingenuity applied when it comes to having context-specific calendars. For instance, during some religious holidays, there may be calendars made with a smaller date scale that serve as a countdown to a day of celebration. For instance, there are chocolate calendars in the Christian tradition that begin on the first of December and countdown the days till Christmas. These calendars typically come in the format of a tray covered in a picturesque cardboard overlay depicting some festive scene. The cardboard has little numbered “windows” that correspond to the date in the month and these can be opened to reveal a small square of chocolate. Another example of a physical context-specific calendar is commonly seen in grade school classrooms. Teachers will track the good behavior and progress of their students with a star calendar. This calendar has days on one axis and students on the other axis. For the days a student behaves or answers questions correctly, a star is placed in the grid square where date and student name intersect.

The passage of time and the relationship of events to this progression is critical to most cultures. Thus, as society has taken steps towards moving into a digital environment, research has been done studying how calendars are used and how digital calendars can be enhanced to either meet people’s functional needs or how they might be presented to people besides the standard layout of days in row. For instance, Neustaedter, Brush, and Greenberg describe their study of family routines to determine calendar attributes needed for planning and coordinating between the different events and responsibilities of the members of the familial unit. They identify necessary guidelines for a family calendar and analyze a series of online calendars according to those principles [3].
In terms of an interest in context-specific calendars, researchers from the University of Maryland in partnership with Microsoft researchers looked into a new calendar interface called DateLens for PDAs back in 2004 when the Pocket PC was a popular item on the market. Their focus was on improving the performance of the calendar in regards to complex tasks like picking good dates for meetings or finding start and end dates for events. Their design used a fisheye distortion technique that worked in a pen-based interaction environment on a small device. The improvement of the calendar was important enough that a big technology corporation saw fit to write a research paper discussing their implementation [4].

Interest in changing tried and true calendars and improving them always comes back to concerns for the end user’s needs and how they may best be met in a new context or improved through a new interface. Trying to bridge the gap between the performance difficulty often found in trying to specify events to a calendar, particular recurring events, the RhaiCAL system introduced a new way of visualizing and interacting with the calendar by having an agent interpret natural language to come up with an initial action to take in the user’s stead in regards to the calendar and then, through a visualization, proposing that action for them to modify or approve based on the accuracy of the agent’s interpretation [5].

Much can be done with a digital calendar in a way that can improve people’s productivity, communication, or utilization of a system. In the case of the digital calendar described herein, the careful design of the calendar can help improve the utilization of some customization features of the portfolio management system it is built into.
Chapter 2

Background

The calendar is implemented within the context of a portfolio management system. Portfolios are the conglomeration of different investment assets held by an investor. The importance of a portfolio mainly stems from the idea of investment diversification. Because securities will typically have varied returns and perform with different levels of success, a mix of different types of assets in the form of a portfolio assures that a smart investor will experience a balanced and stable level of performance for the overall sum of the investments. Even if some assets decline in worth, the hope is that different assets will at the same time be rising in value and through diversification the portfolio will remain financially successful [2].

The task of managing such portfolios often falls to hedge funds responsible for wisely investing their clients’ funds and keeping track of the performance of the resulting portfolios. A hedge fund company employs investment managers who manage the capital through the investment of either borrowed money or the money of their clients. And all of those managers must monitor, update, and keep records for the portfolios they are managing. Essentially these are accounting-like operations applied to a portfolio managing context. There could be many portfolios full of hundreds of diversified investments. An investment manager with fewer clients, working at a smaller hedge fund, may employ the inefficient, but commonly practiced method of storing all requisite information involved with this process in spreadsheets. A spreadsheet application is less expensive than alternative applications available on the market so it tends to be a popular choice for smaller hedge funds.
The specific framework in which the digital calendar is implemented is called MetaBooks, which is an end-user programming environment that can be used to develop specialized accounting systems. It is designed to allow the expected end user to edit and customize the rules that compose accounting-like systems. This expected end user is probably not going to be a trained programmer. Rather, the MetaBooks platform is aimed at accountants in need of specialized accounting systems who have the technical sophistication required for building complex spreadsheets to meet their needs.

The specific MetaBooks-based application that the calendar is currently plugged into is a back-office portfolio management system, aptly named PMS. This application has books that keep track of portfolios of investments as well as the amount each investor has invested in the fund. With a conventional double-entry accounting system amplified by more specialized views, PMS can streamline the workflow for a hedge fund and allow them to make changes to the application as accounting rules evolve.

Given the expected end user, defining calendars has to be easy, and the interface must be comparable to the ways accountants will be using the PMS interface to define new kinds of books and transactions. To understand what interface the end user will be exposed to in PMS, below is an overview of the surrounding system.

2.1 Overview of Design of Surrounding System

The overall design of the whole system and its parts in which the calendar exists can, as brief introduction, best be described by the expected user for the system and the view of the codebase they would be exposed to at that level. MetaBooks is the view of the system that would exist for a programmer interested in maintaining the code and evolving implementation and
functionality of code components. For a programmer, PMS would be considered an example application or test data that could be built from MetaBooks. Only a programmer would have access to the code view, written in Java and Groovy, which is MetaBooks.

The second view is a level higher where the expected end user would be a business analyst who could use a user interface view called FAST to create applications like PMS. This level of interaction is a step above using Eclipse and editing Java project files written in code. Instead, FAST has UI screens that allow for editing metadata, creating new objects, types, and rules. Accounting rules are often subject to change and new things may need to be recorded or kept track of as businesses move into new kinds of practices and technologies. The data gets edited in FAST for the creation of the final application of PMS.

The Portfolio Management System is the web view that would be akin to having a ready spreadsheet with all data types and rules ready for use. PMS is more graphical, easy to use and understand, and has the least technical customization required for the end user.

As all code written for any part of the system, the calendar code runs in a part of the MetaBooks system called Dios. Dios makes the calendar useable by applications like PMS, but it is not truly part of PMS any more than the JVM is part of the Java code that the calendar is written in. By adding a component to Dios, the result is a new definition added to the programming language available in FAST.

It is the calendar’s contribution, instantiation, and usage as a definition in the programming language that is of particular concern. In FAST a business analyst can do a variety of things using Dios definitions to create an accounting application. PMS is created in FAST and has an initialization script that can be run to populate the UI screens in FAST with object models - the DIOS available classes - and instances of those objects. In FAST are built the components
that then make up PMS that gets presented to the user in a more digestible and usable manner via web app.

The way the following sections are described, the main concern presented is how the calendar code will impact PMS as the application presented to the final end user, but this is more a short hand of saying that the main concern is how easy the calendar is to use for the business analyst making changes to an application, like PMS in this case, using the FAST UI-level view of the system. PMS is the specific application created from the accounting context system, and the results, benefits, and difficulties of the calendar trickle-down to the application level. How the code and interface for the calendar is defined in MetaBooks at the programmer level gets exposed to the business analyst using FAST to create the accounting-like application required to meet their business needs. And then, finally, any mistaken schedules, inaccurate fiscal periods, or desires for new patterns get brought up in practice with using the actual application to do business. Hence, the rest of the thesis generally describes the concerns of calendar implementations in terms of PMS.

2.2 Joda-Time

The calendar component in MetaBooks is written in Java, but relies heavily on the Joda-Time library for date operations and formatting. Because dates compose the majority of the calendar’s usefulness, it is worthwhile mentioning it before moving into details of the calendar component. Pre-Java SE 8, the native date operations, formats and functionality were inferior in the standard Java library compared to other libraries and did not meet most needs of people who required temporal awareness in their programs. The popular alternative has typically been Joda-Time, a library that offers easy, extendable date and time operations. Most of our calendar uses
the LocalDate class to create instances representing a day. The LocalDate creates a date, excluding the time, using three arguments - the year, month, and day all provided as integers and ordered from coarser to finer granularity. Lower clock-time granularities are unnecessary in accounting, although Joda-Time does support including the time with the date or only providing the time, which means the calendar could potentially be refactored if ever it becomes necessary to have the time for an event. It would be a relatively easy change to make given how similar Joda-Time classes are to one another and how the arguments are always laid out in descending order, from the coarsest to the finest granularity. However, given that the use cases for the calendar are all on a day-scale, it was deemed unnecessary to use classes like DateTime or LocalTime so as to provide events on the calendar with a higher degree of temporal awareness.

The methods defined in Joda-Time that work on LocalDate classes are what make it possible to compare dates over a particular parameter. Later, the classes that require these comparisons, all instances of the DateSet, will be discussed in greater detail, but the nature of these comparisons includes things like checking whether two dates share the same month, whether a date is between two dates, and finding the number of days between two dates. All these things are available using Joda-Time methods, and all of them return integers, so in Java it is simple to compare, for instance, a LocalDate that has getMonthOfYear() called on it and a stored integer representation of a month that is a part of an instance of a Java class.
Chapter 3

Requirements

When considering a calendar’s requirements, it becomes important to separate the physical and graphical manifestations of calendars, which are the more common formats end users are likely to encounter, from the actual functions a calendar serves. A calendar does not need to be a square numbered grid where a user can pencil information about events. People keep calendars to have some sort of temporal awareness to install predictability of tasks and events into their lives, as well as to store information pertaining to those tasks and events. In addition to the date, time, and name of the event in question, that information may include phone numbers, notes, or, considering a more programmatic view, an object that could be scheduled to execute at scheduled periods. An example of such an executable object could be a system update or automated scripts dealing with testing, cleanup, or initialization. However, though the calendar being described herein is not a graphical one, it still shares many of the same functional requirements.

3.1 Division of Time

A calendar needs to be able to divide time into periods of varying granularity, like weeks, hours, years, or days. Also, when a calendar is more than just a static entity meant to show the passage of time, as in this case, then it needs to be able to pair those periods with events.

3.2 Support Singular and Recurring Events
Some events may be singular in nature, such as special one-time meetings or other spontaneous events that will not repeat with any sort of regularity. There must be a way to add such events to a calendar so they do not repeat. Other events may occur based on a simple schedule, such as updates to the system that may occur every Saturday at midnight. And some events may follow a more complex pattern or a set of rules, as may be seen in more specialized calendars for particular organizations. For instance, in the case of a financial institution that manages portfolios, a calendar specialized for this organization may include a way to schedule events based on business days, which can be defined as “not a weekend” and “not a holiday”, or based on fiscal months and years, where every day is a part of a fiscal month, but each fiscal month’s days will be marked with a different event. To support adding recurring events to a calendar without requiring an end user to go to every day and define it as part of, for example, August’s fiscal month, there needs to be some method to define events that will allow an end user to define the pattern of recurrence for an event.

3.2.1 Querying Events

If given a date, the system should be able to check the date against the pattern of recurrence and decide whether the event is scheduled to occur at that time. The calendar should also be capable of taking an event and calculating, based on the pattern of recurrence, when the event occurred last or when it will occur next. In a financial context, this is particularly useful when dealing with trade dates and settlement dates. When a trade occurs, a trade date must be recorded. However, the settlement date, when the buyer must pay for the securities the seller has delivered, does not have to match the trade date. Often, the settlement date tends to occur three business days after the trade date, although this is not a hard and fast rule. Depending on the
security, the settlement date may be more or less than this and in the case of futures, the settlement date can be as far from the trade date as ninety days or even a few years later. Therefore, a search function that calculates a settlement date some number of business days in the future must be flexible enough to support both near and far time frames.

3.2.2 Flexible Rules for Recurring Events

For recurring events, there are many rule combinations that the system should reasonably support. Commonly schedules define events as occurring every work day, not during holidays, during a season, or on the same day of the month every month. It may be necessary to define an event as “for every other Tuesday” or “the 16th, 21st, and 30th of January every year”. These rules are similar, but contain slight variations and differences in scope that an inflexible system will not be able to support. Some events apply to numerous days or to patterns that are yearly or weekly, monthly or daily. An example of an inflexible system would be one with only hard coded rule options to choose from. Such a scheme would require the programmer to first generate a set of rules containing every reasonable variation and then having the user pick from the set the rule variation that matches the case they want in order to add it to the calendar with an event. Such inflexibility can be seen in designs that heavily rely on multiple choice or drop down menus with the only choices available being those that are preset. Another inflexible design would be one that tries to auto-detect a rule that a user inputs in plain text. Although this method of inputting options would be more flexible to the user, since the input would then be by definition allowed to be less structured, it would also be more error prone. Although people find natural language easier for imparting ideas and communicating, machines need structured
statements, unambiguity, and consistency in order to properly interpret the intentions and instructions of an end user.

The first inflexible system described is completely rigid and more useful to the machine than the user, and the second inflexible system is less taxing on the end user but does not lend itself to accurate and precise results, which are both essential criteria in a system meant to keep the schedule for an end user. These two are on opposite ends of the spectrum, so it is likely that an acceptable solution lies somewhere in the middle, with the user able to customize the system to personal needs and the system able to interpret a consistent method of input.

To handle these nuanced variations, a well thought out solution is required. One that in its usage will not overwhelm or confuse the end user and, in terms of code visible to a programmer, will be understandable and easy enough for outsiders new to the system to maintain. This means that naming conventions are observed, classes are not heavily coupled, the work is decentralized, and other good practices of design are adhered to.

### 3.3 Interface as Models for User Benefit

Of these calendar requirements, handling recurring events is of particular importance and thus must be implemented in such a manner as to be easily accessible to the intended end user. Previously mentioned, the expected end users for MetaBooks are business personnel. By knowing the end user, this informs the interface, much as the application domain of accounting-like applications informs the necessary flexibility of the digital calendar implementation.

If the end user is to be able to use recurring events in PMS, specifying a recurring event must be simple enough for the user to be capable of learning the process quickly and without undue stress. It should not require business personnel to understand how to program or perform
complicated logic beyond that which they would have seen in setting up their complex spreadsheets. All the rules they were used to using need to be simple to implement in the new system. If the learning curve is steep, the end users will have no reason to upgrade to MetaBooks platform. To ensure the learning curve is more a hill than a mountain, the interface needs to be intuitive, and instructions and references for using the interface need to be in a format that the end user will likely have seen or even used before.
Chapter 4

Implementation

Although there is always more than one way to design an implementation, it is generally good practice to avoid reinventing the wheel and to search through professional solutions that are commonly applied to particular problem domain, which in this case concerns calendars and, by extension, scheduling. The calendar needs an interface that will allow users to define the schedules for events, which is simple for singular events and difficult for recurring events that need to be defined by a pattern. To meet this need, a good place to start is to look at professional solutions. Martin Fowler identifies a design pattern for accommodating a flexible, easily expanded system for recurring events. He describes a pattern language that can be implemented with neat, simple, modular code that should be understandable to end users, code maintainers, and the computer system. He suggests first to consider the interface of a schedule, imagining what responsibilities the end user will want to give to the schedule, which may include having it hold information about specific events, being able to run processes at certain times, and returning information about upcoming or past events [1].

Considering the functionality of a calendar, or “schedule”, as Fowler prefers to call it in his specification, he identifies six fundamental components to the design pattern. The Schedule serves as a container for collecting events. This Schedule has an Interface that allows the user to ask it different questions about the events that it is holding. These questions may include whether there are any events scheduled for a particular date, when a particular event will occur again, or what are the details of the event. Another component he describes is the Schedule Element, which links an event to a pattern of recurrence. In terms of components that would make up a
pattern of recurrence, he describes Day Every Month that would take a day of the week and an integer as well as a Range Every Year that takes a day and a month for both start and end dates. With these he is able to construct Set Expressions that represent combinations of temporal expressions, in particular the unions, intersections, and differences [1].

Though the general idea Martin Fowler presents is adhered to in the implementation of the PMS calendar, there are some important differences. Fowler’s motivating example is the occurrence of an event on a particular day of the week in a month. He describes the pattern of having an occurrence every Monday, on the first and third weeks of the month as an example. In the case of an accounting system, the calendar needs to support specific dates, fiscal periods, and non-repeating intervals. In the following sections, the implementation of the PMS calendar is described along with explanations of design decisions and motivations for deviations from Fowler’s solution.

### 4.1 High-Level Overview of Design

The calendar design, as a whole appears in Figure 1 as a class diagram. The Calendar class acts as a holder for all objects that are calendar-related. It is where all the data related to the calendar is stored - mainly, IntervalGenerator instances and CalendarEvent instances. In terms of use cases, the IntervalGenerator can be thought of as being the fiscal periods on an accountant’s calendar, and the CalendarEvent can be considered to be a mix of all other events, be they one-time appointments or patterns for concepts like a business day for Motorola versus a business day for a company in Taiwan. An end user will create the calendar first and then fill it with these schedule-related objects that will express their desired schedule.
As far as functionality goes, there are two main areas - the calendar and the temporal expressions, both of which are explained in detail in the following sections. Before stepping into the details of these two areas, it is natural to begin at the interfaces and notice what functions and attributes make these areas different enough to warrant separate interfaces.

![Class Diagram of Calendar in MetaBooks](image)

Figure 1: Class Diagram of Calendar in MetaBooks
Two main interfaces, one for each area, serve mostly to make provisions for uniform querying. Querying refers to the ability to ask the calendar questions about dates and events or asking instantiated temporal expressions whether a date fits their pattern.

The Calendar, IntervalGenerator, and CalendarEvent are all higher-level classes that answer queries with more complete answers. They all implement the EventGenerator interface which ensures that the end user knows which questions can be asked of the calendar as well as what kind of answer to expect while the IntervalGenerator and CalendarEvent define their own means of answering those questions. What makes their answers more complete is that they return data structures full of either events or dates for events.

The interface for the lower-level classes that extend AbstractDateSet is simpler and less robust, much like the natures of the queries they are meant to answer are simpler. A class that extends AbstractDateSet only needs to take a given date and compare it to its pattern. They all answer with a Boolean yes or no. They do not pass data structures. It is the only question they answer and the rest of the interface for the DataSet classes is only in consideration for end users, to provide a cleaner way of using conjoiners And, Or, and Not. Thus, most DataSet classes are short and simple. The only exception is WeekOfMonth for which no Joda-Time methods are available to simplify calculating which week a given date belongs to, so WeekOfMonth figures out which week a date appears in using its own methods.

It should perhaps be made clear that AbstractDateSet implements DataSet, mainly so as to specify that the interface the end user will be using (DateSet).and(DateSet) actually means a call to And(DateSet, DateSet). This distinction will be discussed later in
regards to temporal expressions in the view of the end user. The classes that are made available to specify patterns of recurrence extend AbstractDateSet, thus overriding the contains() method.

The DateSet is the more important facet of the calendar, because through temporal expressions the DateSet allows for the calendar to be used programmatically with patterns of recurrence being defined much like rules in a spreadsheet, with some small piece of constrained code. Every calendar contains some means of holding dates and events, but the DateSet makes the calendar accessible to the end user, allowing for customized patterns that could vary in complexity and be attached to events that could represent anything from system updates to the business days of funds. Where the higher-level event generating classes can let the end user say what will be occurring, the DateSet will provide the definition for when those events will occur in a flexible manner.

4.2 Calendar and Events

The Calendar class is the means through which all the other pieces get organized and allows for a condensed view of the temporal details of an accountant’s work in one or more separate entities. In other words, an accountant could make either one calendar to hold all the events of every fund or create numerous calendars, each for a different fund. This provides the end user with the flexibility to organize themselves however seems best for the amount of data they must track for each fund and how similar funds are.

The calendar is essentially just a holder for data that can answer specific questions about the data that normally the end user would have to extrapolate with a manual investigation through a list of events. To create a calendar, the end user will create an instance of the
Calendar class by declaring an empty Calendar(). This creates an empty calendar with empty data structures ready to be initialized with the end user’s data.

The first of these data structures is a HashMap that uses String names as keys that map to CalendarEvent objects. The second is an ArrayList that stores IntervalGenerator instances. These two data structures comprise all the data for the calendar. These two kinds of objects are stored separately because both the process for creating these objects and the use cases of when an end user would utilize a CalendarEvent over an IntervalGenerator are different. In the section on Temporal Expressions, these differences will become clear. However, to communicate that difference to the end user, there are more questions that can be asked of a CalendarEvent and there are separate methods to add these objects to the data structures. The addGenerator() adds IntervalGenerator instances to the ArrayList and the addEvent() adds CalendarEvent instances to the HashMap.

```java
//requires import statement for CalendarEvent
CalendarEvent event1 = new CalendarEvent("e1", calendar, dyn, addOneEvent);
calendar.addEvent(event1);
calendar.addEvent("e1", dyn, addOneEvent);
```

Figure 2: Java code showcasing two different addEvent() methods

There are different versions of addGenerator() and addEvent() that either take fully formed IntervalGenerator or CalendarEvent instances respectively, or they take as arguments the data that would be provided to create these instances. Overloading these methods is motivated with considerations for the end user. The end user will write short scripts to create events. Figure 2 shows an example of adding the same CalendarEvent to the Calendar in two different ways. The first way, as seen under the comment, requires the end user to explicitly
create a CalendarEvent instance. This requires an import statement and the CalendarEvent has to identify which calendar it is going to be added to as its second argument. Then, on the next line, it must be added to the calendar. The alternative to this two-step process is to simply give the arguments of the CalendarEvent to the method. This removes the need for creating a CalendarEvent explicitly, does not require the argument that says which calendar the event belongs to since the method can supply that information automatically based on the calendar the add is called on. By having methods for creating object instances, end users can avoid import statements that would otherwise be necessary. Also, they save on the amount of code that needs to be written, since the new keyword and the object name no longer have to be present in their scripts. Without imports and constructors, the end user should have an interface that looks comparable to the scripts they already write for spreadsheets.

Other than adding, getting, and setting, the rest of the methods in Calendar are queries. There are three queries the end user can make to the calendar include - eventsFor(), nextDatesFor(), and previousEvents(). The method eventsFor() allows the user to ask what events occur on a given date. To answer, the Calendar class then iterates through all its data objects and calls the CalendarEvent and IntervalGenerator versions of eventsFor(). Both classes answer with a CalendarEvent if the given date is one that they recognize. The Calendar finally bundles all the events that matched the date into an ArrayList and hand the data structure to the user. The process is shown in the sequence diagram in Figure 3. The iterative nature of the query methods makes it beneficial for the end user to organize their data into several instances of Calendar if the data is not directly related, thus reducing the search space and limiting the amount of data one calendar has to carry. This implementation subtly encourages the end user to be better organized.
The other queries - `nextDatesFor()` and `previousEvents()` do not apply to `IntervalGenerator`. These queries take an event name and a number. The `HashMap` is asked to lookup the event name. If it is there, in the case of `nextDatesFor()` then the `CalendarEvent` instance is told to look at its pattern of recurrence and try dates until it finds the next time the event is scheduled to happen. It finds as many future dates for the event as the user wants and returns those dates in a list. The method `previousEvents()` does the same thing as `nextDatesFor()` except in reverse, trying past dates to get a history of an event’s occurrences.
As for the data itself, `IntervalGenerator` will be discussed in a later section due to its complexity and because it has separate use cases despite the fact that it shares the same interfaces as `CalendarEvent`. From its relatively simple description, the `Calendar` class may not seem robust enough to handle more complex or flexible operations, like being able to execute an update or import data about funds after the workday is over. However, the data classes in the `Calendar` carry with them this functionality in the form of arguments. In particular, the `CalendarEvent` takes four objects. A name that is the key to finding it in the `HashMap`, an `EventGenerator` which is typically a backward reference to the calendar in which the event belongs, a `DateSet` that describes the pattern of recurrence for the event, and a special `MetaBooks` class called `DynamicObject`.

This last argument is the placeholder for more complex use cases of `Calendar` that have yet to be implemented in other parts of the system. Currently, the `DynamicObject` is typically assigned a null value when instantiated in test cases, but the flexibility and functionality it leaves room for would allow users to automate information gathering and initialization. Unfortunately, this feature is not very useable at the moment, because many things that get executed require human intervention and cannot just be automatically scheduled to run.

To explain the flexibility of the `DynamicObject`, it is best to start with a use case. To reconcile the data in the system with the records of different brokers, currently there is a manual process of updating the data by importing files. There are two problems with this that motivate keeping this process manual for the time being. The files that brokers send to be reconciled are typically provided via email. They cannot simply be downloaded via FTP, but even if they were downloaded using IMAP, there is not guarantee for when these files will be sent and the bigger problem remain in that many smaller banks or firms format their spreadsheets so that they are
human readable, with colors and more than one category in a column. These formats are not standardized. So, it is difficult for a machine to automatically read in and find specific data in these spreadsheets so that reconciliation and updates may occur.

Eventually, more businesses may start using systems like PMS rather than spreadsheets, which would mean standardization and files available without searching broker’s email inboxes. Then, it would be more practical to have updates run automatically and necessitate having a process that would act as controller for these updates. This process could wake at midnight, after the workday is over and computing resources are available without interference. It could query a Calendar set aside for updates and reconciliation for events that are upcoming. The process could spawn a thread to start for the event and go back to sleep while the thread executes the event. And it is in anticipation for this need of executing actions that CalendarEvent gets created with a DynamicObject parameter and contains an execute method. Otherwise, its methods are standard - getting, setting, and querying.

4.3 Temporal Expressions

The main focus of Fowler’s paper is the Schedule’s maintenance of “which events (there may be several) occur on which days...with the ‘when’ part delegated to a temporal expression” [1]. The key idea from Fowler is that of the temporal expression, which is much like a Boolean expression in that it allows complex expressions to be defined from simple ones, like Day Every Month, using just And, Or, and Not. In this way, the rule for an event occurring on “the 19th, 20th and 21st of every month” can be defined with “19th Day Every Month or 20th Day Every Month or 21st Day Every Month”. “Day Every Month” is responsible for describing to the computer system that an event will occur on the same day each month. It is not a pattern that can
be chained and so it only needs to take one argument. “Or”, as a conjoiner needed for extending the complexity of a pattern, takes two `DataSet` classes, one of which can also be a conjoiner and it specifies that if any of those three dates are tried against the rule, then the event is occurring. In this way, the user can define complex patterns through chaining. In the case of creating an expression to say that three days in every month of the year will repeat, the example below shows how this could be realized in pseudo code.

```
Or(Day Every Month(19), Or(Day Every Month(20), Day Every Month(21))
```

This pattern is ideal for creating schedules. It is easily extendable, since defining a new pattern simply requires writing a new class to express that pattern. None of the other `DataSet` classes will be affected through the addition of a new pattern since they are all written with good modularity. But from the end user’s perspective, they will be able to create many new kinds of patterns to express an event’s schedule.

Fowler’s design pattern is also relatively easy to understand for an end user. The classes are named so that they describe the functionality provided within the class to let end users know what pattern of days will be captured by it. For instance `DayOfMonth(int)` lends itself to be interpreted by the user as a pattern that, when given an integer, will define a repetition that occurs every time a specific day, like the 15th or the 29th, of any given month is reached. `DayOfMonth` also claims a part-to-whole relationship. A day is the part being defined by the integer and it exists in the whole that is a month. This naming convention should remove the confusion of whether it is the day that repeats in the pattern or the month. And, given that there is only one parameter, it cannot be both month and day being repeated. Since Fowler’s pattern of defining recurring events meets the requirements of both flexibility and functionality, the
question then becomes how much will need to be customized to support the given system
requirements and context specific needs.

Modeling Fowler’s implementation for temporal expressions, smaller and more numerous
dateSet classes serve a similar function as Fowler’s SetExpression, Day Every Month, and
Range Every Year descriptions. Having numerous types of patterns for recurrence with
descriptive names should simplify the interface to the end user because the code mimics an
English description of the pattern of recurrence. The purpose of the dateSet classes is to
capture a pattern and be able to say whether a date fits the pattern, returning true if it does and
false if it does not.

This return of true and false is done through the contains() method, which takes a
date and is responsible for comparing that date to the pattern of its instance. The contains()
method is part of the interface for all classes that implement dateSet, but each class has its own
definition for whether a date is contained in its pattern of recurrence or not. For instance, when
an instance of the DayOfMonth class is invoked, it is given a day that is meant to define the
pattern belonging to that instance. So, DayOfMonth(20) will result in an instance of
DayOfMonth being defined as occurring every month on the twentieth. The contains()
method simply does a comparison between the given date, which may be whatever date the end
user wishes to check, and the date that the instance of the DayOfMonth pattern holds, which in
this case is the twentieth. If the given date also occurs on the twentieth, the contains()
method returns the Boolean value true. For all simple patterns that only take one argument, the
contains() method always just matches along the parameters that the dateSet stipulates
should be the same. If a date is given to a DayOfWeek instance, then the day of the week of the
given date has to be the same and nothing else. If the given date and the DayOfWeek instance
pattern both occur on Thursday, then it does not matter if the years or the months do not match, the response from contains() will be true.

The logical specifiers - And, Or, and Not - also have the contains() method, but they define the functionality of the method with Boolean expressions rather than date comparisons. Since the constructors of logical specifiers take other instances of DataSet, which have contains() methods that return true or false, the logical specifiers just perform the correct logical operation on the results returned from singular specifiers. To illustrate this with the And DataSet, if And is given a DayOfMonth and DayOfWeek, the contains() method in And will call DayOfMonth.contains() and DayOfWeek.contains() with the given date and do the Boolean and on the results, thus returning true or false for whether a date fits a particular pattern of recurrence.

In terms of what patterns the end user can use to define schedules, there are seven singular specifiers available that fall into two categories. The first category includes DayOfWeek, DayOfMonth, WeekOfMonth, DayOfYear, and MonthOfYear. These are specifiers that have a clear cut pattern and will be recurring periodically. With the DayOfWeek, DayOfMonth, DayOfYear specifiers, an event can be defined as recurring on a particular day every week, such as every Monday of every week, a particular day in a month, such as every 15th of every month, or a single day in the year, like Christmas.

With the WeekOfMonth specifier, an event can also be defined as recurring on particular weeks of a month of the five weeks possible in most months. Weeks are counted in terms of Saturdays. The first Saturday is found. Any day before the first Saturday is considered a part of the first week, even if the week is incomplete or only contains one day. Days that trail the last Saturday are considered a part of the last week of the month. In February there is occasionally
only four weeks, as can be seen in the year 2015, when the first of February is on a Sunday, and the twenty-eighth is on a Saturday. This is an uncommon case for February and the other months always have five weeks, though they are incomplete weeks with less than seven days. Thus, one through five are all acceptable inputs when creating a pattern using WeekOfMonth.

Another singular specifier, MonthOfYear, handles the possible recurrence for when events occur in a particular month in the year, such as every February. However, some events that need to be placed on a calendar may not follow a recurring pattern or may simply be singular events that will not repeat. The end user will still want to place these on a calendar, and thus there must be a means to add such dates. This second category contains the final two singular specifiers which cover this case.

Add and NonRecurringRange specifiers are similar, but useful in different contexts. The NonRecurringRange specifier allows for an event to span a number of days between a start and an end date. Festivals, school semesters, and sports seasons are examples of ranged events that may change beginning and end dates from year to year and generally occur in a consecutive period of time. A semester does not stop after a week only to be reconvened a month later. Allowing for a range of dates to be specified reduces the effort for a user in inputting dates. There is no need to add each day of a semester manually nor does the user need to use some complicated compilation of rules that will need to be thrown out after the non-repeating semester is over. However, in the case that there are peculiar events that occur on nonadjacent days but are still considered the same event, for instance if a user wants to define holidays, then the Add specifier allows an end user to input a list of dates for an event.

The holiday example is of particular interest in a Portfolio Management System since one of the common cases for patterns of recurrence in accounting and business contexts is defining
business days. Typically, a business day is more easily defined by what it is not. For instance, a
business day is \textit{not} a weekend and it is \textit{not} a holiday, but it \textit{is} everything else. With \textit{Add}, an end
user can easily make a list of all the year’s holidays, which may be country specific and occur on
different days every year, such as Easter which is a religious holiday that can occur anytime
during the spring and often has strange rules for its occurrence. \textit{Add} also increases flexibility. A
reasonable way to define holidays would be to use a number of \textit{DayOfYear} specifiers conjoined
together. However, by including Easter in this fashion would require that the rule be tossed every
year as the date for Easter would change. Because of \textit{Add}, it is possible to have separate
categories of holidays - those that repeat and can use \textit{DayOfYear}, and those that are easier to put
in a disposable list, to be renewed every year as the dates change.

The singular specifiers are useful, but can only be utilized effectively if there are
conjoiners available to combine patterns. In order to say that an event is \textit{not} something or that it
is a combination of somethings, logical operations are required, much as they are needed in
Boolean expressions.

To perform the temporal expressions as defined in Martin Fowler’s paper, PMS has three
logical operations - \textit{And}, \textit{Or}, and \textit{Not}. The \textit{And} and \textit{Or} both take two arguments, both of
\textit{DateSet} type, thus allowing for complex patterns with chained expressions. The results of \textit{And},
\textit{Or}, and \textit{Not} follow the truth table for their logical counterparts in Boolean expressions. Since
the singular specifiers all have a way to return whether or not a given date fits into their pattern,
the logical operations can return the results of whether all parts of the whole pattern is true for a
date.
For instance, the `DataSet` defined for business days is shown in the test case above. First, to avoid a long slog of code, the desired dates for holidays are defined using the `LocalDate` class from Joda-Time. The holidays are then given to the singular specifier `DayOfYear` to indicate that these dates will be recurring every year and strung together with the conjoiner `Or` since a holiday can occur on any of those dates. Another `DataSet` is then defined for the weekend, which generally includes only Saturday and Sunday. So, finally, the business day `DataSet` can be constructed from the `DataSet` classes for weekend and holiday with the use of `And` and `Not`. Thus, a business day is not a weekend and it does not occur on Independence Day, New Year, Christmas, or Christmas Eve. Here, again, the consequences of using `Add` versus `DayOfYear` can be noted. This pattern can be tested with using Christmas of another year.
Still using the example pattern for business days defined in Figure 4 when a calendar is trying to discover whether a date belongs to an event, it calls the `contains()` method that all `DateSet` classes implement. Given the date December 25, 2018 to check against the pattern, the date is applied to all parts of the pattern. Since this date occurs on a Tuesday, the `DayOfWeek` instances return false - the date does not fit their respective patterns for Saturday or Sunday. `Or` returns false, since both of its singular specifiers return false and `Not` flips the result, since a business day should not occur on a weekend. The date fits the first half of the rule, but the `And` ensures that the date must meet both requirements of being neither weekend nor holiday. The second part of the pattern has Christmas 2014 included in the pattern with `DayOfYear`. Since `DayOfYear` recognizes the date given to it as being repeated every year, it will return true. `Not` will overturn this result and the pattern will fail. Thus, December 25, 2018 is not a business day.

However, the pattern defining holidays in the business day pattern could have been written using `Add` rather than `DayOfYear`. This would look as shown in Figure 5.

```
//holidays
final LocalDate christmas = new LocalDate(2014, 12, 25);
final LocalDate newyear = new LocalDate(2014, 1, 1);
final LocalDate independence = new LocalDate(2014, 7, 4);
final LocalDate eve = new LocalDate(2014, 12, 24);
ArrayList<LocalDate> holly = new ArrayList<LocalDate>(){{
    add(christmas);
    add(newyear);
    add(independence);
    add(eve);
}};
DateSet holidays = new Add(holly);
```

Figure 5: Java code for creating a pattern of one-time holiday events using `Add`

With this implementation, when the date reaches the second part of the pattern, the `Add` looks for Christmas 2018 in its list, and it does not find it. The given date has to match the listed
date exactly, so with one occurring in 2018 and the other in 2014, the result is a return value of false - the date does not fit the pattern. Not overturns this result, and And will succeed, thus informing the end user that December 25, 2018 is a business day. Considering that many holidays typically remain as either business day or not business day from year to year and continue to occur on the same date, an end user may find it more desirable to use the recurring pattern for DayOfYear rather than relying on Add lists. With this example it is apparent that the system has the flexibility to allow for either means of specifying a pattern.

Together, singular specifiers and logical specifiers make it possible to identify the days in which an event is occurring, but this is not enough when considering the context of calendars meant for accounting, finance, or portfolio management. In particular, the contains() method only says whether an event is occurring, but when an event occurs every day of the year, it is not enough to respond to the user with a true or false, since the answer will always be true. Instead, the end user will be interested in the name of that event. Thus, to handle this case, there is the IntervalGenerator.

4.4 Fiscal Periods

Fiscal months and fiscal years divide time into periods. Sometimes fiscal years are the same as the calendar year, but often they start in the middle of a calendar year or even in the middle of a calendar month. With the Dataset classes mentioned so far, there is no way to define a continuous stream in which there is always an event and the only variant between one day and the next is its name that changes depending on the date. Such a scheme is necessary because there is never a date on the calendar that does not belong to a fiscal month. There are no loose days and every calendar date has exactly two fiscal events - one for the year and one for
the month. Thus, the importance is not in whether a date has a fiscal period event, since the answer for a fund’s calendar is always true. Rather, it is important to know exactly which fiscal month or year a given date belongs to.

To solve this issue, the IntervalGenerator provides a means of creating separate events for each fiscal period, each with its own unique name. It only creates events, always of NonRecurringRange, when an end user calls eventsFor() and provides the proper name to the given fiscal period. If a fiscal month for a company begins on the 15th of one month and ends on the 14th of the next month and follows the naming convention of defining the dates in that period as belonging to the month in which the period starts, then the IntervalGenerator must respond to the date September 13th with a generated event that has the name “August Fiscal Month” because it belongs to the period that began in August and ended in September. This calculation of the name and auto-generation of events rather than just checking for the existence of a fiscal event requires a different interface, thus IntervalGenerator implements the EventGenerator rather than the DateSet, and instead of answering if it contains() a date, it passes along the eventsFor() a date, which includes fiscal period information.

To instantiate IntervalGenerator, the end user needs to provide three arguments. The first is simply a name for the instance, which can be any arbitrary String. The name serves mostly as a way for the calendar to find events when eventsFor() is called. It is an overloaded method that can be called with either just a date or an event name and a date. In the case of IntervalGenerator, the calendar checks its ArrayList for generators that have the name of interest. This is faster than calling eventsFor() with only the date because otherwise, every IntervalGenerator must be called to provide events for a date, which they may or may not have. So, to that extent, it benefits the end user to name generators descriptively to indicate the
pattern it represents, which will typically be fiscal month or fiscal year so that when they later query the calendar, they can use the `eventsFor()` that takes a `String` as an argument.

The next argument to `IntervalGenerator` is a `DataSet`, so a pattern of recurrence. This pattern defines an infinite stream of start dates for which the `IntervalGenerator` then can generate an infinite number of `NonRecurringRange` events to describe one distinct fiscal period with its own unique name. Usually this `DataSet` will be simple, comprised of either a `DayOfMonth` pattern if the generator is for fiscal months or `DayOfYear` if the generator is for fiscal years.

```java
private CalendarEvent newCalendarEvent(LocalDate start, LocalDate end) {
    DynamicObject result = null;
    try {
        result = (DynamicObject)objectGenerator.evaluate(start, end);
    } catch (UserException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } catch (EvaluationException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    if(result == null){
        return new CalendarEvent(this.name, this, result, new NonRecurringRange(start, end));
    }else{
        return new CalendarEvent((String) result.getProperty("name"), this, result, new NonRecurringRange(start, end));
    }
}
```

**Figure 6: newCalendarEvent() method called by eventsFor() in IntervalGenerator**

On a call to `eventsFor()`, the generator will take the given date and return an `ArrayList` of `CalendarEvent` objects - or, most likely, just one object. These events are created in a call to `newCalendarEvent()` that takes the start and end dates and creates a `CalendarEvent` that has `NonRecurringRange` as its pattern of recurrence. This process is
necessary so that the user can be given a single list of events of a matching type. The method 
`eventsFor()` will usually originate as a call to the calendar, and the calendar then will call 
`eventsFor()` in the `IntervalGenerator` as well as asking `CalendarEvent` objects in its 
`HashMap` to check the date against their `DateSet` patterns to see if they are occurring. So that 
`IntervalGenerator` can answer with a matching type as the `CalendarEvent`, it creates a 
singular event that by definition does not repeat, which is acceptable since a specific fiscal period 
is constrained by a start date and end date much as a `NonRecurringRange`. The 
`NonRecurringRange` event that the `eventsFor()` in `IntervalGenerator` creates also 
takes the third parameter of the `IntervalGenerator`.

The third parameter, `objectgenerator`, is of type `Evaluable`. In PMS, an 
`Evaluable` object can use the method `evaluate()` which will turn it into a `DynamicObject` 
as seen in Figure 6 and passes the result as the `DynamicObject` part of the event constructor. 
This `DynamicObject` is expected to be a `Function` object, as seen in an instantiation of 
`IntervalGenerator` in Figure 7.

```groovy
def intervals = new IntervalGenerator("Test", new DayOfWeek('mon'), 
    new Function('start, end: Types.FiscalMonth.newInstance([start: start, end: end])', 
        ObjectStore.instance))
```

Figure 7: Groovy code for defining an `IntervalGenerator`

The `Function` class is responsible for calculating the `DynamicObject`. Essentially, it is 
a wrapper for a Groovy closure, which is an open, anonymous block of code that can be passed 
as an argument to functions or constructors much like could be more commonly seen in scripting
languages like JavaScript. **Function** knows the source code, unlike a closure which needs to be calculated. A closure cannot be saved to disk, but a **Function** will take the closure and write the closure code as a **String** onto disk. To retrieve it, the closure is simply reconstructed from the **String**. The **Function** class given in Figure 7 will be the typical format that the end user will be able to model from test cases. The rule for naming a fiscal period or month is written as code for a closure and given to **Function** as a **String**. The second argument of **Function** is simply a directive for what to store it as. Thus, depending on the date being queried, the date becomes an argument to whatever rule is associated with the instance of the **IntervalGenerator** - its third argument - and a name is calculated. In Figure 6, if the **IntervalGenerator** was created with **Evaluable** not equal to null and it was evaluated - likely a **Function** object being evaluated - then the result of that evaluation, which is a **DynamicObject**, is asked to provide its name in creating the **CalendarEvent** return type, and this will be the specific name for the date’s fiscal period, such as “April Fiscal Month,” however this flexible **Evaluable** argument to **IntervalGenerator** leaves room for possible future use cases to be more easily implemented. Flexibility is important, but not more so than heeding user preferences and behaviors when designing a system, or even a component.

### 4.5 Design Decisions Concerning End Users

In the context of a financial application meant for accountants to utilize in practice, the interface of the different components has to be match in both style and simplicity. People are resistant to change, particularly when their livelihood is on the line. Weaknesses in one component can reflect poorly on the whole of the system and discourage adoption of new methods. So the calendar, much as any other component of PMS, must not be a source of
frustration, confusion or slowdown for neither the user nor components that utilize or interface with the calendar.

Therefore, after designing the interface for the calendar the first time, it was necessary to revisit the implementation for flaws and analyze the tradeoffs of designs and deciding implementations based on which criteria was deemed more important. Two concrete examples stand out in the interface as having required these sorts of decisions.

```java
Dataset business_old =
    new And(new Not(new Or(new DayOfWeek("sat"), new DayOfWeek("sun"))),
    new Not(new Add(holidays)));

Dataset business_new =
    ((new DayOfWeek("sat").or(new DayOfWeek("sun").not()).and(new Add(holidays).not()));
```

**Figure 8: Old and new syntax for defining a business day pattern**

These changes to the calendar had a direct impact on the interface that is exposed to the business analyst editing PMS features through the FAST GUI. Figure 8 shows the syntax used in the initial implementation of the calendar, stored in the variable `business_old`. Below it is the revised interface, stored in `business_new`. They are equivalent patterns for defining a business day as neither a weekend nor a holiday. They both employ the `Add` which takes the variable `holidays`, an `ArrayList` of `LocalDate` instances. They both employ `And`, `Or`, and `Not` in the same way. However, in `business_old`, the logical operations require that an instance of themselves be created each time. Thus, the keyword `new` appears more frequently and to a non-programmer this may seem strange and could hinder debugging customization efforts if they forget to include them. Having so many keywords also reduces readability, although readability is already poor considering it is easy to lose track of the balance of parentheses and the way the
logical specifiers appear on the outside and take parameters within means that they must be read from the inside out for the conceptual meaning to be clear. And it is important to understand the conceptual meaning of the patterns that have been defined quickly. The accountant customizing PMS with coded patterns has the primary task of keeping accurate track of portfolios and capital changes from their investments. The programming is just a means to an end and it needs to require minimal effort.

The new interface improves on the old in that it puts the conjoiners Or and And between things that are being conjoined so that the code can mimic English speech and the Not negators are appended to the Dataset classes they are meant to negate, which allows the user to first code a case, think through the components needed in it and evaluate it for whether it should be negated or not at the end. The coding process becomes more natural and there are less parentheses, they are less nested, and the logical specifiers no longer require the new keyword, reducing the debugging effort. End users should not even be tempted to place the new keyword in front of logical specifiers in this new format because it is so different from the singular specifiers. As to using new in front of singular specifiers, it should be more understandable to the end user because in natural English the accountant could think of new DayOfWeek(“Sunday”) as saying “I want to define a new pattern that includes the day of the week Sunday”.

However, if the new way of defining the Dataset rules is still too burdensome to the end user, there are two more alternative ways of defining a business day that use a Groovy feature that allows for the programmer to extend a Class that belongs to someone else to include more methods. For instance LocalDate belongs to Joda-Time, but it can be extended to include a dayOfMonth() as shown in Figure 9.
The result of this code is that the new keyword required of instances can be entirely removed, as can the import statements that would normally be required for a script using `DateSet` classes. The code in Figure 9 starts with the class that is being expanded and is followed by the method being added. Then, in brackets is the code that occurs in this new method. The result in the syntax for giving the pattern for a weekend is as follows.

```
('sat'.dayOfWeek()).or('sun'.dayOfWeek())
```

String is a Java class that will not require an import statement in PMS and so the String ‘sat’ has a method defined in MetaBooks being called on it much like native String methods would be called on a string, such as `substring()` or `length()`. And, if the business analyst is more comfortable with mathematical expressions and `DateSet` patterns being defined more like actual formulas, the other alternative that further simplifies the syntax but removes the natural language is as follows.

```
('sat'.dayOfWeek()) | ('sun'.dayOfWeek())
```

These different methods of writing `DateSet` patterns will all work without error, but the last two, where imports are no longer required and logical operands - the bar for Or as well as ampersand for And - are the examples mainly seen in the documentation. The difference in these simplified formats and the reason for having both be available to the end user is due to the
difference in people. Some people are more comfortable with using natural language and will find the first simplified method more useable. Others may like defining what are essentially formulas using the more computational annotation of bar or ampersand. It is better to accommodate possible user preferences and considering what different people may find intuitive than it is to constrain them into utilizing a single method that they may find difficult to use.

Beyond changes to ways of defining Dataset patterns, the second major change to the interface also was through making some assumptions about human intuition. In the initial calendar, all the Dataset classes uniformly took a single LocalDate argument. Thus, to define a DayOfWeek pattern, the end user would have had to pick a date that happened to be on a particular day of the week. That same date could have been given to any of the Dataset singular specifiers without error. It would be easy, in a moment of inattention, to substitute one singular specifier for another, particularly since they are named similarly. Also, the end user would have to look at an actual calendar to debug a mistake in a pattern that included WeekOfMonth or DayOfWeek since they both necessitate knowing the temporal attributes of a date.

The revised version made it so singular specifiers take more descriptive arguments. A DayOfWeek takes a String representation of a weekday and accepts any String for a day of the week as long as the first three letters correctly spell the beginning of a day of the week. So, DayOfWeek("sat"), DayOfWeek("SaTuRdAy"), and DayOfWeek("Satur") are all acceptable ways to define a pattern of recurrence for Saturday. Thus, there is more flexibility, the natural language sense of the rule is retained, the operations are shorter to instantiate, and the singular specifiers are harder to mistake for one another in the face of distraction since they take different arguments and have different maximums, so most mistakes will cause errors.
<table>
<thead>
<tr>
<th>Uniform Arguments</th>
<th>Intuitive Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DayOfWeek(new LocalDate(2015, 3, 31))</code></td>
<td><code>DayOfWeek(&quot;tues&quot;)</code></td>
</tr>
<tr>
<td><code>DayOfMonth(new LocalDate(2015, 3, 31))</code></td>
<td><code>DayOfMonth(31)</code></td>
</tr>
<tr>
<td><code>DayOfYear(new LocalDate(2015, 3, 31))</code></td>
<td><code>DayOfYear(new LocalDate(2015, 3, 31))</code></td>
</tr>
<tr>
<td><code>WeekOfMonth(new LocalDate(2015, 3, 31))</code></td>
<td><code>WeekOfMonth(5)</code></td>
</tr>
<tr>
<td><code>MonthOfYear(new LocalDate(2015, 3, 31))</code></td>
<td><code>MonthOfYear(&quot;March&quot;)</code></td>
</tr>
</tbody>
</table>

Table 1: DateSet singular specifier constructors reformatted for end users

In Table 1, all the singular specifiers are shown presenting equivalent patterns in both the new and the old method of instantiating them. To create such diverse patterns as “occurs on Tuesdays”, “occurs on the 31st of every month”, “occurs every March”, and “occurs in the fifth week of the month” the singular specifiers could have taken the same argument in the previous calendar implementation. Often times, uniformity is desirable and remains in the naming conventions. However, at some point the threshold is reached where too much uniformity makes it difficult to identify the differences between items. There is a tradeoff in changing the arguments, since the new way requires a user to remember the type of argument each singular specifier takes. The design decision to choose the new way was motivated by the consideration that debugging is more difficult and frustrating when the mistake is something the compiler cannot catch, such as thinking March 31st was a Monday rather than a Tuesday and a silent error like this can cause critical mistakes in accounting whereas a syntax error will not silently pass
and should be preferable to an end user, even if it initially requires looking up what the arguments need to be for a given singular specifier.

Another kind of design decision is extension of code functionality or additional flexibility. After looking at the rules and trying to instantiate different patterns, it became apparent that one subtle pattern was not possible. It was easy to create a pattern that occurs on the first or the fifth of the month, but not to create a pattern for the fifth day from the end of the month. Given that DayOfMonth takes an integer that represents the day of the month on which the pattern is to repeat, there was no way to say some fixed time from the end because months do not uniformly end on the same day. The fifth day from the end of the month of April is the 26th because April ends on the 30th. In March, the fifth day from the end is the 27th since it is a 31-day month and for February, it can be either the 25th or the 24th depending on whether it is a leap year or not. To cover the case where a pattern occurs some number of days from the end of the month, DayOfMonth was revised to also take negative numbers. The constructor therefore checks whether the day given as a pattern of recurrence is negative. If it is, then when contains() is called with a day, it looks at the month the date occurs in, uses Joda-Time functions to calculate the end date of that month and then subtracts the correct number of days to check the given date against the pattern date.

The changes to the interface described are not tremendous. But just a few small differences can be the difference between adopting a new system and sticking with a tried and true method of accounting.
Chapter 5
Performance

Testing can function as more than just a means of validating the correctness of the code. When test cases reflect real-world use cases, they can serve as a model for how an end user will utilize key features. By benchmarking these tests, the code maintainer can identify which operations are slow in performance. Then an analysis can be made of the tradeoffs between having slow but simple code or increasing the complexity for better performance but at a diminishing return on effort.

In the case of PMS, there are a few primary use cases for the calendar that would interest a given accountant using the system, such as the user can create a calendar, the user can add events with different patterns to the calendar, the user can find future events and ask the calendar for all events that will occur on a given date.

When dealing with this implementation of a calendar, finding the next case of an event is slow because for every event there is not a pre-calculated set of dates. Instead there is a \textit{DataSet} associated with an event. That \textit{DataSet} only answers to the query \texttt{contains()}. Therefore, when an end user wants to find the \texttt{nextDatesFor()} an event or the \texttt{previousEvents()}, the code for these operations has to call the \textit{DataSet} with a chronological set of date suggestions and collect the dates where responses come back positive, up to the number of next or previous dates the user wants.

Finding next and previous events this way is simple to understand and easily implemented, but due to its iterative trial-and-error means of arriving at the answer, it is clearly a target for possible improvement efforts because of its inherent inefficiency. However, before
effort should be taken to change the process, first it is good practice to actually decide whether the operations are as expensive as they seem.

To decide this, a common solution is to try benchmarking the code, which generally requires writing code that utilizes these possibly expensive operations and timing the execution. Two ways were tried in experimenting with which way was best to benchmark the code. The first was using the Java system call to get the current time, `System.currentTimeMillis()`. This call was made right before the code of interest and right after. Then the difference was calculated to get a sense of how long the process took. This way is rather primitive but it is accurate enough and allows for fine grained control of which instructions get captured. However, printing the results to a file became problematic. At first, the redirect for output to be printed to a file was coded in a setup method in the JUnit file with the tests in question. The setup method is purposed for creating the environment that all the tests share. The expected behavior of placing the output redirect in the setup method is that a file could be declared and all the output in every method in the test class would append its results to that file’s contents. However, only print statements within the same scope as the redirect instruction were printed to the file. Test cases in other JUnit test methods all printed to the console instead of the file. Also, the information provided by just using `currentTimeMillis()` is not very robust and any other details about what is being run need to be written into the print statements by the programmer. It is not a significant task, but it is tedious. The solution would have been to declare the redirect in each method, but then an alternative was found by way of JUnitBenchmarks.

JUnitBenchmarks is a set of extensions that create GC-monitoring, time variance measuring performance micro-benchmarks from JUnit4 tests. There are numerous benefits to using these extensions that make them a preferable alternative to the primitive print redirect.
They record execution time averages and calculate the standard deviation as well as record garbage collector activity, thus giving the programmer an idea of the waste being generated by the tested code. Also it is much easier to implement and data visualizations can be automatically generated with only a pair of simple annotations. All that is needed to turn JUnit4 test cases from a JUnit test class into benchmarks is an additional field written within the test class, as shown in Figure 10.

```
@Rule
public BenchmarkRule benchmarkRun = new BenchmarkRule();
```

Figure 10: Annotation for adding benchmarking to test cases

This line in the test code will indicate to JUnit runners that benchmarking code should be added to each of the tests. Or, instead of using Figure 10, the programmer may also simply have the testing class extend AbstractBenchmark. Either way, the tests will be run as benchmarks, not only once, but multiple times in the effort of better estimating the average execution time of the methods being tested. To have greater control over how many times benchmarks are run and if any warm-up rounds should occur first, annotations can also be used to specify these details or even to provide visualizations [6, 7].

Having decided on JUnitBenchmarks, the test case file is written in Groovy to accommodate a more realistic setup for the system environment the calendar component is expected to run in. The organization of the test code has one parameterized method act as a calendar creator that the setup code calls with the specifications for the data needed to create either a test of a large, medium, or small dataset with events of different kinds of Dataset.
patterns or IntervalGenerator instances. This creation method has event-generating loops nested inside an outer loop, which simply acts as a faster way of controlling the size of the datasets in case they needed to be twice as big. Since DataSet patterns are constrained by possible values - March 40th is not a valid DayOfMonth pattern - the inner loops use the mod operator to ensure that the varied inputs to different DataSet instances are real possible inputs. To skip a category, for instance to exclude IntervalGenerator instances from the dataset, the loop control variable, defined as an argument to the parameterized creation method, can just be set to zero. Thus, any kind of dataset can be created and tweaked just from the setup method where different kinds of calendars with varied workloads get created.

The benchmark tests focus on the two trial-and-error methods - eventsFor() and nextDatesFor(). The previousEvents() method is left out because it is essentially equivalent to nextDatesFor() except in reverse so the performance is expected to generalize. The eventsFor() method is tested for a calendar that contains only events, only generators with DayOfMonth patterns of recurrence, only generators with DayOfYear patterns, and all four types in equal measure.

Since CalendarEvent and IntervalGenerator both override the eventsFor() method in a meaningful and unique way, it is worthwhile to know the difference in their execution time. They both run eventsFor() on the same dataset, a list of 15 dates. The number of dates tried is so small because in order to be able to compare the performance, the same size dataset must be used in both a test of CalendarEvent and IntervalGenerator. Because IntervalGenerator instances with DayOfYear can take a long time to finish when it is being run ten times to find an average, the dataset is kept at a reasonable 15 elements.
To choose workloads for testing, it is preferential to have information about real world workloads. In the absence of real world workloads, an educated guess needs to be made. Considering that PMS is intended for small to medium size hedge funds, it seems fair to think that if they are working at a small scale, they might have 50 patterns and events defined, 500 at a medium, and 2000 at a large scale. Companies can each employ their own rules about non-business day holidays and each of them has fiscal months and years, many of them possibly unique. Also, using the use case described earlier of scheduling updates or importing datasets after the working day is over, companies may all have different patterns for when they update data or want to download records for reconciliation. When there are millions of possible investments that could be made, it seems reasonable to expect that small and medium hedge funds may have as many as 2000 patterns in calendars. A neater summary of the `eventsFor()` benchmarks that were run is described in Table 2.

The benchmarks are run in Eclipse on a Windows 8.1 64-bit operating system with an Intel(R) Core i7-4700MQ CPU running at 2.40GHz and with 16GB of RAM. The code is run sequentially on a single thread with five warm-up rounds that are not measured and ten rounds that are measured so as to reach a consensus on the average run time of methods. The two methods are run separately.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Calendar</th>
<th>Dates to get eventsFor()</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOYIntGenLrg</td>
<td>2000 DayOfYear of IntervalGenerator</td>
<td>15 LocalDates</td>
</tr>
<tr>
<td>DOYIntGenMed</td>
<td>500 DayOfYear of IntervalGenerator</td>
<td>15 LocalDates</td>
</tr>
<tr>
<td>DOYIntGenSmall</td>
<td>50 DayOfYear of IntervalGenerator</td>
<td>15 LocalDates</td>
</tr>
<tr>
<td>DOMIntGenLrg</td>
<td>2000 DayOfMonth of IntervalGenerator</td>
<td>15 LocalDates</td>
</tr>
<tr>
<td>DOMIntGenMed</td>
<td>500 DayOfMonth of IntervalGenerator</td>
<td>15 LocalDates</td>
</tr>
<tr>
<td>DOMIntGenSmall</td>
<td>50 DayOfMonth of IntervalGenerator</td>
<td>15 LocalDates</td>
</tr>
<tr>
<td>CalEventsLrg</td>
<td>250 of all 8 kinds of DataSet (simple and complex) CalendarEvent = 2000</td>
<td>15 LocalDates</td>
</tr>
<tr>
<td>CalEventsMed</td>
<td>62 of all 4 simple DataSet and 63 of all 4 complex CalendarEvent = 500</td>
<td>15 LocalDates</td>
</tr>
<tr>
<td>CalEventsSmall</td>
<td>6 of all 8 kinds of DataSet (simple and complex) CalendarEvent + 2 complex = 50</td>
<td>15 LocalDates</td>
</tr>
<tr>
<td>BothLrg</td>
<td>200 of all 8 kinds of DataSet (simple and complex) CalendarEvent + 200 of both Month and Year IntervalGenerator = 2000</td>
<td>15 LocalDates</td>
</tr>
<tr>
<td>BothMed</td>
<td>50 of all 8 kinds of DataSet (simple and complex) CalendarEvent + 50 of both Month and Year IntervalGenerator = 500</td>
<td>15 LocalDates</td>
</tr>
<tr>
<td>BothSmall</td>
<td>5 of all 8 kinds of DataSet (simple and complex) CalendarEvent + 5 of both Month and Year IntervalGenerator = 50</td>
<td>15 LocalDates</td>
</tr>
</tbody>
</table>

Table 2: Description of benchmarks by types and numbers of IntervalGenerator and CalendarEvent objects and number of dates being searched
5.1 Results and Analysis

The results of the benchmarks for `eventsFor()` are shown below. `DOYIntGen` and `DOMIntGen` refer to the calendars which only contain `DayOfYear` and `DayOfMonth` patterned `IntervalGenerator` instances, respectively, thus covering the case of only fiscal years and only fiscal months. Both refer to a calendar that contains a smaller number of every type of `IntervalGenerator` and `CalendarEvent`. And `CalEvents` are the calendars that contain only instances of `CalendarEvent` in different patterns of singular and logical `DateSet` patterns. `Lrg`, `Med`, and `Small` are a reference to the calendar element size, 2000, 500, and 50 respectively. The results of running the benchmarked tests can be seen in Figure 11.

![Figure 11: Average time in seconds for each calendar to find 15 LocalDates](image)

It is no surprise that the larger datasets take longer to execute, however, one can notice from Figure 11 that executing `eventsFor()` on `CalendarEvent` data is inexpensive even in the case of a large number of `DataSet` patterns stored in the calendar. This may be due to the fact that `IntervalGenerator` iterates through all the dates between the provided date and the pattern for start dates to find the nearest start date, which may be very far off from a provided date when a `DayOfYear` pattern needs to be matched. After it finds a start date, `IntervalGenerator` goes on to create `NonRecurringRange` events. Meanwhile, a `CalendarEvent` only calls `contains()` on the pattern its holding. The same data from the graph is laid out in Table 3.

<table>
<thead>
<tr>
<th>method</th>
<th>Average time [s]</th>
<th>StdDev</th>
<th>GC time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOYIntGenLrg</td>
<td>8.95</td>
<td>0.31</td>
<td>1.71</td>
</tr>
<tr>
<td>DOMIntGenLrg</td>
<td>5.05</td>
<td>0.09</td>
<td>0.33</td>
</tr>
<tr>
<td>DOYIntGenMed</td>
<td>2.49</td>
<td>0.08</td>
<td>0.08</td>
</tr>
<tr>
<td>BothLrg</td>
<td>1.65</td>
<td>0.06</td>
<td>0.07</td>
</tr>
<tr>
<td>DOMIntGenMed</td>
<td>1.52</td>
<td>0.06</td>
<td>0.07</td>
</tr>
<tr>
<td>BothMed</td>
<td>0.61</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>DOYIntGenSmall</td>
<td>0.58</td>
<td>0.1</td>
<td>0.03</td>
</tr>
<tr>
<td>DOMIntGenSmall</td>
<td>0.43</td>
<td>0.03</td>
<td>0.03</td>
</tr>
<tr>
<td>BothSmall</td>
<td>0.36</td>
<td>0.05</td>
<td>0.01</td>
</tr>
<tr>
<td>CalEventsLrg</td>
<td>0.08</td>
<td>0</td>
<td>0.02</td>
</tr>
<tr>
<td>CalEventsMed</td>
<td>0.08</td>
<td>0</td>
<td>0.02</td>
</tr>
<tr>
<td>CalEventsSmall</td>
<td>0.07</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3: Exact numbers for all ten runs of benchmarks, standard deviation, and garbage collection

The `DayOfYear IntervalGenerator` costs the system more time than any other method in garbage collection and its standard deviation is much higher. This is probably due to
the iteration from given date to start date. If the given date is close to the beginning of a pattern, there are fewer dates to compare to the start date; otherwise the whole year may need to be iterated over. Considering this is such an expensive operation in comparison to the CalendarEvent version of eventsFor() and it is mostly because of creating objects and executing Evaluable, one possible solution that is implemented to counter the issue is a cache.

The cache is really just a data structure in the IntervalGenerator which maps a date that could be given as an input to eventsFor() to the CalendarEvent result that it returns, thus skipping the evaluation of Evaluable and the object creation. The results, displayed in Table 4, show that adding a cache vastly improves performance, particularly in the case of a DayOfYear IntervalGenerator. Not only does the cache increase performance, but it is also a small, understandable change, easily maintained or enlarged to keep track of more dates, which may be necessary if DayOfMonth is discovered to be a more popular method. The DayOfMonth IntervalGenerator hardly changes with the addition of the cache due to some amount of thrashing. Not many dates being tested occur in the same month, so the cache misses more frequently.

<table>
<thead>
<tr>
<th>method</th>
<th>Average time [s]</th>
<th>StdDev</th>
<th>GC time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOMIntGenLrg</td>
<td>4.95</td>
<td>0.09</td>
<td>0.36</td>
</tr>
<tr>
<td>DOMIntGenMed</td>
<td>1.51</td>
<td>0.05</td>
<td>0.07</td>
</tr>
<tr>
<td>DOMIntGenSmall</td>
<td>0.44</td>
<td>0.06</td>
<td>0.06</td>
</tr>
<tr>
<td>DOYIntGenLrg</td>
<td>3.39</td>
<td>0.21</td>
<td>0.89</td>
</tr>
<tr>
<td>DOYIntGenMed</td>
<td>1.05</td>
<td>0.03</td>
<td>0.04</td>
</tr>
<tr>
<td>DOYIntGenSmall</td>
<td>0.36</td>
<td>0.03</td>
<td>0.02</td>
</tr>
</tbody>
</table>

Table 4: IntervalGenerator benchmark run results with cache implemented
The performance of IntervalGenerator with the cache is not fast, but the operation completes quickly enough that no further solution is needed. If, however, these performance issues were present in the eventsFor() in CalendarEvent, then a more complex solution would have to be implemented. This priority difference stems from the intuition that business analysts will not have very many IntervalGenerator objects with DayOfYear patterns, or even DayOfMonth patterns, on their calendars. The use cases pertaining to IntervalGenerator are simply less frequent. Many calendars will likely only have two or three IntervalGenerator objects, just to describe fiscal periods. The rest of the objects in a calendar are likely to be of type CalendarEvent, and of these there may be thousands. The eventsFor() method is very common and with the makeup of calendars consisting mostly of CalendarEvent objects, the priority is to make the common case fast. And in this case, it is sufficiently fast to satisfy a user and seems to scale well in terms of performance on larger versus smaller calendars.

In the case of nextDatesFor(), the method is only meaningfully utilized by CalendarEvent instances. It works by trying dates against contains() in an event’s DataSet pattern. Before it moves into this iterative process, it requires the end user to provide the name of the event so that it is always just one event that is being looked at. Though its description sounds slow, surprisingly it fares quite well in performance. The size of the dataset given to the event is 18000 lookups, because of 40 event names to find five nextDatesFor() that occur after 90 different start dates. The number of iterations that nextDatesFor() is called is 18000 times and the results are listed in Table 5. It only takes about a minute to do this and the size of the calendar does not change the results because the Calendar uses a HashMap to store
CalendarEvent which means there is no iteration required prior to an event calling nextDatesFor().

<table>
<thead>
<tr>
<th>method</th>
<th>Average time [s]</th>
<th>StdDev</th>
<th>GC time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>nextDatesJustEventsMedium</td>
<td>58.06</td>
<td>29.3</td>
<td>20.63</td>
</tr>
<tr>
<td>nextDatesJustEventsLarge</td>
<td>57.36</td>
<td>29.74</td>
<td>17.36</td>
</tr>
<tr>
<td>nextDatesJustEventsSmall</td>
<td>58.26</td>
<td>30.53</td>
<td>18.4</td>
</tr>
</tbody>
</table>

Table 5: Average results of ten runs of nextDatesFor() benchmarks

Considering that the nextDatesFor() method will usually be used on patterns for getting the next trade or settlement date, which in most cases will occur in three days’ time, a small amount of inefficiency is tolerable since there is little cost to performance. The only case nextDatesFor() could take longer than calculated by the benchmarks is in the case of futures, which can occur years later. This will mean that the code will iterate a long time to find the next dates. However, it is unlikely that a business analyst will want to search for futures in this manner. It is a rare operation and as such the possible drop in performance when futures are being calculated is outweighed by the cost in refactoring effort, understandability, and maintenance.
Chapter 6
Conclusions and Future Work

The context an application will be used in often informs the design in some way. Machines, devices, online applications, game systems, and software in general are rife with digital calendars. However, there are so many because they are used for different things and, as a result, employ different designs. If one calendar could serve every need, applications would not ship with temporally aware feature sets. A programmer cannot sit down to design something without knowing the context in which it will be used, even if it is something well-known, like a calendar.

In the context of the financial industry, the dates for an event are of particular importance because money can be made or losses incurred due to inaccurate event scheduling. Most accountants rely on spreadsheets in which to keep their data and so most of the information is manually tracked for settlement and trade dates and fiscal periods or business days for different funds, both international and national. MetaBooks and PMS provide an alternative solution to the tracking needs of accountants and business analysts, but for them to adopt a new system all parts of the system must be usable, flexible, and easy to understand. Thus, the context heavily informs the type of calendar that should be available to the end user. In this thesis, the reasoning for choosing Martin Fowler’s pattern has been thoroughly explained as have the means in which the calendar was implemented.

Through the process of implementing the calendar, it became clear that design patterns vary significantly from one another and some fit better than others to a problem domain. It is best to consider professional solutions and then customize them to the context rather than reinventing
the wheel. Design is an iterative process and that understanding of the project and its requirements evolves with time and trial-and-error. And even with a professional solution to model from, interfaces need to match their users, just as performance needs to match expected workloads. Just knowing the problem domain does not necessarily mean accurate knowledge about the nature of the users and their expected workloads.

For the future, there is room to experiment with this implementation of the Calendar to improve performance or simplify the interface. For instance, one experiment could be to make singleton constants for the immutable objects in the calendar, so that instead of new DayOfWeek("Sunday") the user could avoid creating objects with a singleton DayOfWeek.Sunday. Or, if nextDatesFor() is found to be a more frequently used method, it may be beneficial to change its serial, trail-and-error implementation to a lazily evaluated infinite stream of dates. And beyond functionality and performance, MetaBooks may experience internationalization. The Calendar may then be extended for other languages and include new, more culturally-informed patterns of recurrence.

With a flexible initial implementation, the Calendar may easily have its patterns of recurrence extended, some new queries can be added, objects may become executable, and more accurate information about workloads may result in changes to methods that had previously seemed uncommon and were allowed to remain inefficient. Even for such a simple component in a much larger system, a lot can change. The only sure thing about this implementation of the calendar is that it will change with time and that, as long as it is in use, it will never truly be finished.
References


