PP: Smalltalk Mini Project
NAME: Alexander Brandborg
STUDENT NUMBER: 20136225
STUDENT MAIL: abran13@student.aau.dk
STATUS: Program lives fully up to requirements and is runnable
SMALLTALK SYSTEM: Pharo Development 1.1.1

Overview

For my solution I have created the CAL-abran13 package, including five new classes. These can be seen in the inheritance diagram in appendix A, where they are marked with blue.

An instance of Appointment includes a String object to describe it, as well as an AppTimespan object. The latter represents the temporal span, from when the appointment starts, to when it ends.

An instance of Calendar includes an OrderedCollection object known as components. As required, we may add both instances of appointment and calendar to this collection. This poses a challenge, when iterating over the collection. To overcome this, I implement the composite design pattern. This requires that the two classes share an interface, which is enforced by having them inherit from the abstract Component class. This class describes the methods that each subclass needs to implement to live up to the interface. As both classes share an interface, we can safely send the same message to every object in a calendar's collection. As we iterate over a calendar's components, we choose the method implementation through dynamic dispatch. We say that a method implemented on both classes is a polymorphic operation.

AppDateAndTime inherits from Pharo's DateAndTime class. It is very similar to its parent, but does not allow for seconds or nanoseconds to be set through the constructor methods on its metaclass. According to the project description, a point in time should only be described down to the minute, while it is described down to the nanosecond in DateAndTime. The restriction is enforced by overriding constructor methods on the AppDateAndTime metaclass, which are inherited from the DateAndTime metaclass. The new implementations require that seconds and nanoseconds are set to 0.

AppTimespan inherits from Pharo's Timespan class. The difference from its parent, is that its starting time has to be an AppDateAndTime object. Again, this is done to ensure that we do not worry about seconds or nanoseconds. I enforce this by overriding some of the constructor methods on the AppTimespan metaclass, which are inherited from Timespan's metaclass. The new implementations require that the starting time is of type AppDateAndTime.

In Appendix B I depict code metrics for my solution. These were extracted using Pharo's introspection features. I show metrics for both classes and metaclasses, as I have added constructor methods to metaclasses throughout my solution in order to ease instantiation.

Reflection

In this section I will reflect over the areas requested in the project description.

Object Oriented Relative to Functional

In general, when tackling a problem with the functional approach we define a new function for each functionality. For each data-type that the functionality is required to work on, we add another case to the function. In the object oriented approach, we define the same functionality several times, having an implementation spread out on each data type.

I worked very much according to this idea in my object oriented solution, by spreading out functionality across different classes. This in general made the code more readable, as every method was kept short. In my functional solution some functions grew larger, as I had to handle several data-types in the same function. I generally tried keeping my functions small, when working in Scheme, as large Scheme functions are very difficult to comprehend. This was however an active effort, while this came more naturally when working object oriented.

A downside is of course that if I have to add new functionality, I have to alter code in all classes, which require that functionality. With the functional approach, I will just have to add a new function. The functional approach of course has more difficulty, when new datatypes are introduced, but I think there's a bigger chance of new functionality being added in the case of the calendar system. As an upside, if I need to extend the functionality of the Appointment and Calendar classes, I can enforce such an update by adding an abstract method to the Component class.

In my object oriented solution I also rely more on the code made by others. Reusing code is not unique to object oriented programming, but pushing responsibility onto other objects and methods is very much central. Coding according to this mindset, I used some of Pharo's predefined classes to aid my solution, especially when handling time. In turn, this ensured that my object oriented solution became much smaller and more readable than the functional one. A downside to using code made by others is of course that you place some of your trust in another programmer and their code. You must also have a good understanding of this code before using it. This is however a small price to pay, as to not reinvent the wheel.

Object Oriented and Functional as compared to the requested solution
One of the biggest differences between the functional and object oriented solution is how traversal across calendars occurs. In the functional solution this required a recursive function that ran across the contents of a calendar. Each element traversed was actively categorized as either an appointment or calendar, as to figure out how to handle it. Having to implement the composite design pattern in this manner was rather cumbersome. It was much easier to implement in the object oriented solution, as explained in the overview. Using dynamic dispatch, I do not have to worry about the element type, when iterating over a calendar and sending messages to the elements I find.

In contrast to the functional approach, the object oriented way of programming allows us to have a state. This of course removes some of the restrictions applied to the functional solution. However, it also makes testing more difficult. The output of each method not only depends on its input, but also on the messaged object. I feel that this required me to concentrate more, when testing parts of the object oriented solution. An upside is that the object oriented solution creates fewer data instances. In the functional solution, any update to a calendar's contents required a new calendar to be set up. In the object oriented solution, this simply requires that the object's internal state is updated. This of course leads to the issue of dealing with objects as reference types. Here we must be sure to copy objects when working with them, if we do not with to change the state of the original.

In my functional solution I implemented predicates as lambda functions, which could be send to higher order Scheme functions. This was doable as functions are first class citizens in Scheme. In Smalltalk, instead of lambda functions we have block closures. These objects work similar to lambda functions, but in addition to some code, they also include a copy of the environment, in which they are instantiated. This lets us write more flexible predicates. Yet, again we have the issue that the output of a block not only depends on input, but upon its internal state.

A positive about the object oriented approach is its focus on reflection. In my project I have only used some introspection to extract code metrics. Yet, the ability to perform intercession at runtime to change meta objects adds a a lot of flexibility.

Dynamic Typing Vs. Static Typing

Dynamic typing means that we do our types checks at runtime and types are associated with values. With static typing we do type checks at compile-time and types are associated with variables.

The general upside to dynamic typing is that it allows us to be more flexible. We do not have to define the type of each variable we need to use, as the types are associated with values.

Both Scheme and Smalltalk have dynamic typing. In both cases I was able to very quickly set up and run my code. Yet, with both I also got frustrated, when I accidentally send the wrong type to a function or method. This was caught at runtime and not compile time as no type checking was performed before run. This in turn makes code written in dynamically typed languages more fragile, as we do not have the safety net provided by a static type system. In addition, programs written in statically typed languages will generally run faster, all other things being equal, compared to a dynamically typed program. This is because type checking does not occur at runtime.

To ensure that a dynamically typed program is stable, there is a bigger requirement for testing. Working in Scheme I thought that testing was rather difficult with the tools available. Yet testing is central to working in Smalltalk. Smalltalk is not just a language after all, it is also a virtual machine with a debugger and test runner. In Smalltalk, tests are just classes, which inherit from TestCase, and each method defined within such a class is a bunch of assertions. There is also support for set-up and tear-down of the testing environment before and after testing of a certain class.

To produce quality programs, testing needs to be central to development when working with dynamically typed languages. The upside is that this may force a greater focus on testing, which in turn leads to programs that are more stable than their statically typed counterparts, which rely more on their type system. It may after all not only be type errors that we find during testing. The issue is of course that if we do not place testing front and center, we may end up with very unstable programs.

Benefits From Modern Object Oriented Languages

Smalltalk is in a way very minimalistic. It tries to keep the amount of constructs to the minimum. This is also why Smalltalk has a "pure" object model, where, almost, everything we work with is an object. The benefit of this is that it is very easy to get started with. Everything is an object, even classes, and most everything happens through message sends. In addition the code implementing every class that we interact with is available to us. Yet, in the end I may want to switch out some of this minimalism for constructs found in modern languages.

For instance, like many modern languages with object oriented features, DART has a a constructor defined for each class it sets up. This means that there is one way, and only one way to instantiate objects of a certain class. In Smalltalk we may provide several constructor methods upon a class's metaclass, to allow us to set up objects with certain parameters. Yet, this is not required. I had a big issue with this, when creating the AppDateAndTime and AppTimespan classes. For each of these classes, I had to override the constructor methods on their metaclasses, as these allowed for seconds and nanoseconds to be set. It would have been much easier, if each class just had a single non-inheritable constructor. This would have given me greater control, over how others are able to instantiate my classes.

A related issue is the lack of access modifiers as we see them in languages like Ruby. In Smalltalk, every instance variable is private and each method is public. If we want a private method, we can place it within a "private" protocol. This is however more like asking the programmer to be super special nice and not call the method, rather than restricting them. In relation to my solution, the lack of access modifiers means that you can add seconds and nanoseconds to an AppDateAndTime object through the "second:" and "nanosecond:" methods. These are in a "private" protocol, but a less scrupulous developer may still use them. On the other hand, having no access modifiers does give developers more freedom, to do as they please.

From what I can see, modern languages do not support reflection to the same degree as Smalltalk. Some languages like Ruby do have some introspection features, but not the entire package as presented in Smalltalk. As stated, I have not used reflection much, but it seems to lend a lot of flexibility to programmers. For instance, we can use reflection to change the inheritance tree at runtime, leaving us with dynamic inheritance. Yet, I am not sure if this sort of power should be given to all programmers, as it could potentially lead to code that is very difficult to comprehend and debug.

How To Implement the Solution in the Actor Paradigm in Erlang

Erlang follows the actor paradigm and is all about concurrency. Instead of working with threads and semaphores, we run isolated processes. Each process may spawn and send messages asynchronously to other processes. In addition a process may alter behavior at runtime.

Processes choose how to respond to messages, which is very object oriented. However, in Erlang processes are implemented as functions, not objects. A function usually has some local variables, as well as a receive expression containing clauses. Any message received by the process is run through the receive expression's clauses to look for a match. If a match is found, the code contained in the matched case is executed.

To implement the calendar system in Erlang, we will have to define a function for each new class found in the object oriented solution. These will work as classes to spawn processes of certain types. A function will contain the same variables as the corresponding class. In addition, it will contain a clause in the function's receive expression for each method defined on the old class. The case for the findAppointments method will look as follows:

{findAppoinments, Predicate, Caller_PID}

To match this, another process must send along a message that consists of a tuple. This tuple should contain the findAppointments atom, an anonymous function to be used as predicate and lastly the ID of the caller.

At a pattern-match, we can choose to execute the findAppointments code directly in the process, and then message back the answer to the caller through the Caller_PID, we received. Alternatively, we could delegate the task and spawn a new process specific to the method in question, which could execute the method code. This would allow the original process to work on other messages until the result has been calculated and messaged back.

A calendar process will have a list of all calendar and appointment processes, which it contains. This allows us to use the processes, when answering method messages. In addition, by linking the calendar to its contents, we can have them terminate along with the calendar, when time comes. Each user may have a specific user-process set up. Through this process, a user can message the calendar and appointment processes to call methods. When the user process receives a result back, it has the task of displaying the results. This allows us to support several users, as well as separating the task of calculating and displaying data.

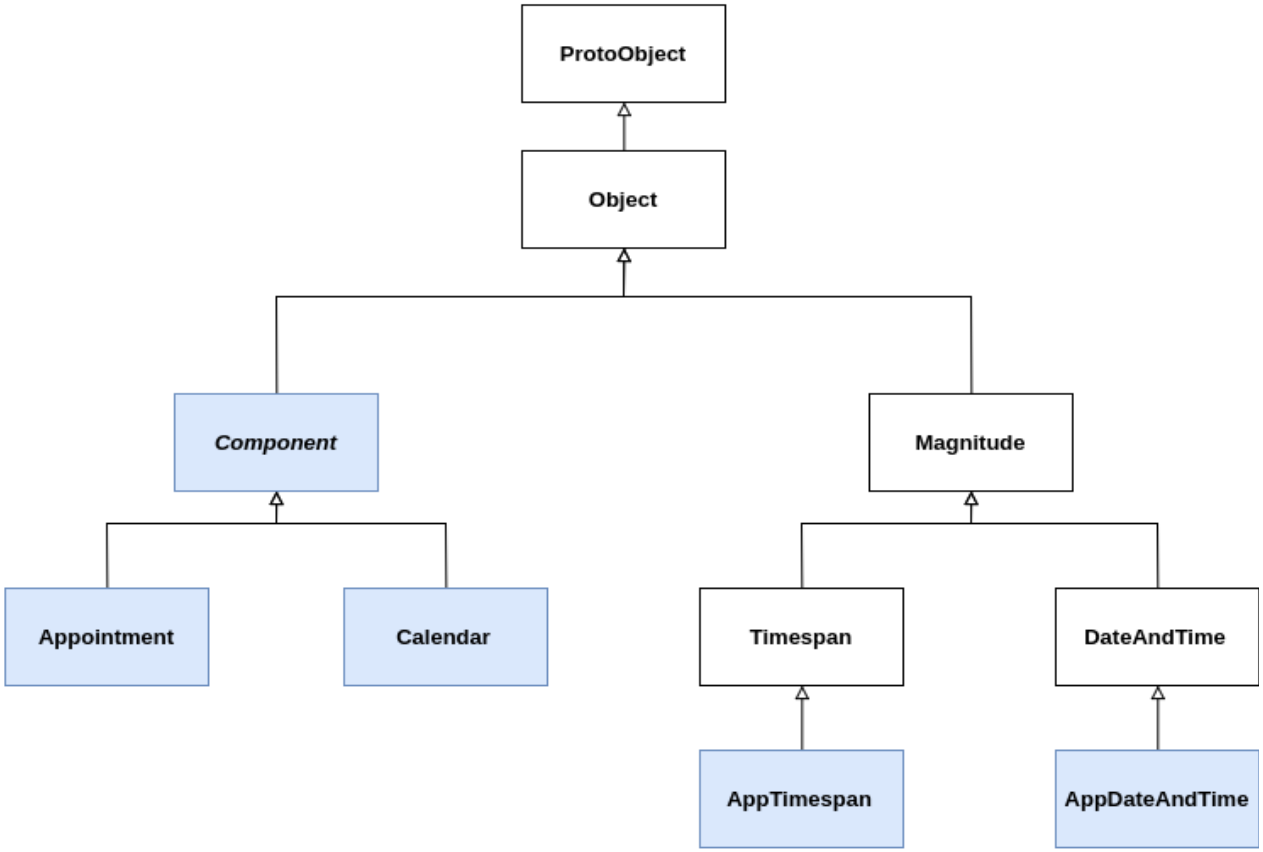
Sources Of Inspiration

I have used no other sources of inspiration for my solution other than the curriculum and the classes found in Pharo.

Appendix

A. Inheritance Diagram

Inheritance diagram showing the classes involved in the solution. Classes marked with blue are created for the solution. White classes were already present in Pharo.



B. Code Metrics

Standard code metrics for the classes exclusive to the solution. Extracted using Pharo's introspection features. Inheritance depth and number of subclasses are the same for both classes and metaclasses as their inheritance trees are parallel. Total lines of code is the sum of lines of code in both class and metaclass.

Class / Metric	***AppDateAndTime***	***App-Time-span***	***Comp-onent***	***Appoint-ment***	***Calendar***	***Total***
Inheri-tance depth	4	4	2	3	3	NaN
Number of new instance methods	2	2	4	13	19	40
Number of new class methods	7	6	0	2	1	16
Number of new instance variables	0	0	0	2	1	3
Number of new class variables	0	0	0	0	0	0
Total sub-classes	0	0	2	0	0	2
Total lines of code	58	59	12	69	107	305