Effective FPL – Feature-Programming Techniques

Prelude

I actually planned on writing this article for quite a while now, but I never got around to it (being too busy writing cool new features for the CAESES users), so I'll give it a try now and see where it ends...

One key to fully exploit CAESES' power are Features. Features are basically small custom made programs that can be executed inside of CAESES and allow the user to change CAESES' behavior in a very flexible (and almost omnipotent) way to suit their needs (I will now start to use "you" instead of the generic, but kind of clumsy, "the user" because I expect readers of this article to be CAESES users - if you are not, you should be!).

I intentionally called Features "programs" because after all, that's what they are. This also means that programming is involved and you have to use (and learn) a programming language. I hope this doesn't scare you too much. I'll do my best to take away your possible fear: for basic - but already quite powerful Features - you don't have to know a lot more than you already know (since I expect you to already be a CAESES user) and CAESES already has built-in functionality to get you started.

Although the programming language we will use has no official name (that I am aware of), I like to call it the Feature Programming Language (FPL - hence the title of this article). Actually, since I wrote the documentation of the *FFeatureDefinition* type, you will even find that name inside of CAESES. Syntactically it is based on the programming language C++ (or more correctly C++'s origin C). So if you have any experience in a programming language that's syntactically based on C (like Java, C#, PHP, Perl, Javascript and others) you do have a head-start (and if you don't, don't worry, it's not that hard).

Over the years, the FPL has grown to a very mature and rich programming language that offers many of the features you may know from other languages. Also, the execution speed of features has been continuously tweaked (though we are still striving for even better performance, of course), so I can honestly say, if performance of your CAESES models (i.e. update times) is a major concern for you, it is highly unlikely that it's caused by the execution overhead that is introduced by Features (of course, what you are doing in those Features is a completely different story - so yes, Features may take a long time to execute if they implement complex algorithms, for example, but the time penalty is not caused by Feature-technique itself).

If you do suspect that your Feature(s) are a performance bottleneck in your project, you can always contact us and maybe whatever you are doing in your Feature is so cool and - most importantly - relevant to other users that we just implement it directly as built-in CAESES functionality.

By the way, in order to find out if your suspicion is correct, you can utilize the cool new profiling mode, that allows you to keep track of how long each object takes to be updated. There's also a Feature-specific profiling functionality that will be covered at the end of this document.

In this article I will try to show you, how to use the FPL effectively. I will also try to keep those of you reading that are scared of the word "programming", but I cannot guarantee that I won't lose you. I

apologize beforehand and hope that this doesn't cause you to abandon Features forever. Just keep in mind that they are mighty powerful and may fill a hole in CAESES' functionality that you can't (or better don’t want to) live without!

I want to stress that my intention is not to provide you with a complete reference to the FPL. I think that the basic syntax control structures are already explained pretty good and thoroughly in the documentation of the *FFeatureDefinition* type. The available commands of the class library that is available in CAESES are (more or less) well explained in the whole documentation that comes with CAESES, as well. Trying to put all of this into a single document would a) be a huge book that hardly anybody would enjoy reading and b) would probably already be outdated by the time such a book would be done, due to new features and small changes that come with each new CAESES version. Instead, we are constantly working on updating and improving the documentation that is available inside of CAESES to be able to offer you the most complete and up-to-date information for the version you are working with. We are currently working on an improved search for the documentation that makes it easier for you to locate the content you are interested in. I know that that part is not always very easy and we need to get better at it. Also, especially when you have specific questions that you just cannot seem to get answered in the documentation, it's always a good idea to visit our forum where not only members of FRIENDSHIP SYSTEMS are very actively involved, but also some of our experienced users offer great advice. Finally, the articles that are published on our website contain great advice, too - not only regarding Feature-programming, but also regarding modeling and general use of CAESES. So it's always worthwhile to stop by.

My goal here is not to force stuff on you. If you are an experienced Feature-programmer you have certainly adapted some customs that you use over and over again in several of your Features. I am just trying to present you some new tools for your tool-box that a) I consider to be the most efficient for certain tasks and b) may not be known to you, yet. I know, "if you are a hammer everything looks like a nail", but sometimes it might just be useful to have a pair of pliers or a screwdriver as well. Actually, it might even be that your current hammer works better for driving that one screw than my proposed screwdriver does. In that cases it would be wonderful if you could drop me a note (a.bergmann at friendship-systems dot com).

This brings me to the topic of future revisions to this document. The FPL is in a constant state of change. We (and you) come up with new ideas every once in a while that we'd like to have implemented for the FPL. Luckily, we do not have to run through a long process to do so (unlike other programming languages). So, if we add new features to the FPL I will try to keep this document updated. Also, if you (or anybody else, including me) find an error in this document, I will also update it. If I do so, I will keep you informed in my developer blog which can be found by clicking the link in my signature of my forum posts.

Again, I want to stress that your input is not only very appreciated but is also very helpful for me and all the fellow Feature-artists out there!

A word on the title: it's actually a rip-off from the very good books of Scott Meyers about the C++ programming language. So, if you have any affiliation to C++, you should really read his books (Effective C++, More Effective C++, Effective STL and Effective Modern C++).

Words

As in every document with a technical topic, I will use certain terms that might not be self-explanatory if you are not an expert in the particular field. Here's a quick overview about frequent words or terms that I may use that fall into this category.

Class and object - both of these are very common in object oriented programming languages (which the FPL is). I will try a short introduction for those that are still scared of the whole "programming" part: Basically, everything in CAESES (a curve, point, surface ... heck, even a double value) is an object. An object, in general, holds data and offers ways to manipulate that data (a manipulation could also be just reading the data - so manipulation in this case does not necessarily mean changing the data. A class is basically the description of an object. Objects are also referenced as "**instances**" of a class. This is the programmer's approach to model the real world. Following this metaphor, everything that exists in the real world is an object with certain attributes (their "data") and those objects can interact with each other. One simple example would be that a human is an object with the attribute "hand" and a pen is an object with the attribute "ink". Those two objects can interact by applying the operation "write" of the human that uses the mentioned attributes of both of the objects.

Inheritance and base-class - classes can be hierarchical related to each other. This means that certain classes (or better the objects of that class) share certain attributes, but may differ in other. Coming back to our example, all humans share a lot of common attributes, but men and women do have certain attributes that are not shared. So, in this case, human would be the base-class and man and woman are the **derived classes**.

Basictypes – There are some fundamental data types, most notably numerical types or strings. In CAESES those types are called basictypes.

Feature vs **feature** (very subtle difference, I know), the latter refers to functionality of CAESES (or any other software, for that matter), while the former refers to the CAESES feature "*FFeature*" (see what I did there ;-)?) which is the whole subject of this article.

FPL - as already mentioned - refers to the programming language that is used to write Features.

Feature - we already resolved the ambiguity between "feature" and "Feature". There's something else, though. Features are split into two parts. The *FFeatureDefinition* and the *FFeature*. All *FFeatures* are based on an *FFeatureDefinition*. The definition is basically the description of what the *FFeature* does (that description is done using the FPL). In general, when I talk about Features, I mean the description, so the *FFeatureDefinition*. I like to refer to actual *FFeature* objects as "instances" of the definition. So, unless I explicitly talk about *FFeature* or "instances", I am talking about the *FFeatureDefinition*. Actually, if I get carried away, I might also use the term “definition” which is another synonym for *FFeatureDefinition*.

Command - In CAESES the term command is used as a generalization of the - in programming languages - more common terms "method" and "function". "Methods" are operations that are invoked on objects. Those operations may or may not alter the state of an object. "Functions" are invoked in the "global scope". They (usually) do not change the state of the system, but they might change the state of an object that is passed to them as an argument. I must admit, that there may be functions that change the global state of CAESES, but those functions are hidden deep down below in

the caves of our office and you should never be in the situation that you need them. If you are (there is a lot of stuff under the surface) you can, though. But, if you happen to find some of the hidden things by accident, let me tell you: stuff that is hidden (i.e. cannot be found in the documentation or in auto-completion) should not be used unless our support team tells you to do so. It will most likely lead to "undefined behavior" if you do not follow this advice.

Members / Attributes – Those two are synonyms and describe the data of CAESES objects. For Features this describes all objects that are created within a Feature

Arguments – This is the input data an object works on. For Features this can be configured in the Feature-editor on the “Arguments”-tab.

GUI – Short for “graphical user interface”

Parameter vs. Arguments – Commands (i.e. functions and methods) usually get some input data they work with. That data is passed to the command as “arguments”. The command itself takes “parameters”. So, at the call size (i.e. the place where a command is called) the data that is passed into the command is called argument, and the command itself takes those arguments as values for its parameters. Here’s an example: The command `echo(FString str)` takes a parameter of the type *FString*. When calling that command `echo(“Hello World”)`, we pass the string “Hello World” as an argument to the command and thus set the value of the parameter `str` to the given value.

There will be other special words or terms that I'll use, but they only make sense if they are explained in a context, so I'll do that as we stroll along.

Interlude

Wow. This much text already and we haven't even started, yet. Maybe it would be better to turn this into a book instead of an article. Good idea, by the way. Book sales generate money and opposed to others that are thinking otherwise, I have to say more money less problems (but maybe I just didn't reach their money-level yet – NO – I am sure that I haven't).

Introduction

Please take the introduction to the introduction from the prelude!

The FPL is an object oriented programming language. I will not try to define the term "object orientation", Wikipedia knows all about that. You can find my try of a very short explanation in the “Words” section.

Once a Feature is written, there are two ways to use it (it's possible to only allow either one of those two ways, but I'll get to that later). A Feature can be used as a custom type (a "class"), that is called persistent use, i.e. you create an object; or just as a script which would probably be called "macro" in other applications (transient use).

A transient feature may execute commands on its arguments, may execute global commands, and it may create new objects (after all, creation commands are nothing but global commands). Objects that are created during the transient execution of Features will be part of the model after the execution has finished. There is another scripting mechanism in CAESES which is based on fsc-files. The difference between those two mechanisms is that transient Features allow to implement more advanced algorithms using control structures, while fsc-files allow to directly manipulate the model.

Persistent Features are a lot more powerful. They allow to create objects with arbitrary behavior that are tailored just to your needs. They can even offer custom made commands to alter their data (I'll tell you how to do that later). That's the kind of Features I'll focus on here, but most of the stuff, I'll cover is valid for transient execution as well (I'll try to remember to point out things, that'll just not work with transient Features or could be a source for trouble).

Structure of this article

I'll structure the individual chapters of this article into what I call "items". The basic idea is to actually offer one main concept per item. Sometimes, I must admit, I might wander off a little and present additional thoughts that are related to each item's basic idea, but, in my opinion, do not deserve an individual item. I want to stress that the order is not a ranking regarding importance. So, if you feel like you already know everything about a certain topic, just skip to the next one.

Also, I'd like to apologize in advance if the order of some of the items may seem odd to you. I am doing my best to put them in a somewhat meaningful order that allows you to follow the advice given in each item without having to jump to downstream items. Maybe I will not always succeed with that, though. I will give it a small revision, once I'm done, though.

General remarks

There's more to a Feature than just code. Let's take a quick walkthrough to the other aspects that define a Feature:

General Settings

The probably most prominent aspect of the general-tab of the Feature dialog is the possibility to give a name to a Feature. This defines not only what is shown in the object tree, but also the name of the creator command for instances of the Feature, which will be "`fp_`" (for "feature persistent") followed by the name given here (I will talk about creator commands more detailed in Item 1 - for now it is just important that they exist). Additionally, the name of the *FFeatureDefinition* also defines the name of the type of instances of the definition. Many times this is rather unimportant, but it does become relevant if you ever have the need to cast an object to a Feature instance or if you want to use instances inside of `foreach`-statements (both, casting and `foreach` will be discussed later on). The spelled out type of an instance is "*FFeature::<name of the definition>*. So, let's say, for example, that your definition is called "BestFeatureEver" instances will have the type "*FFeature::BestFeatureEver*".

There are more general settings, though, and overlooking them or even purposely ignoring them may have huge impact on the behavior and even the correctness of your Feature and instances that are based on it and since I keep on seeing errors in the general configuration of Features, I'll try to make you aware of those potential traps, so read on.

Recreate on update

There are many misunderstandings and due to that many misuses of that option. In general, enabling this option means that the whole code that you defined for your Feature will run every time an instance went out-of-date and receives a request to update. Considering performance this may seem scary and might lead to the impression that you definitely want to disable it. However, most real world Features will not behave correctly if this is disabled.

Let's consider what happens when disabling this option. The creator-code will run once, when the instance is initially updated. All objects that are created inside the Feature will be created (based on the current state of the arguments - this is the important part) as "members" of the Feature. All dependencies (between the members and to the arguments) will be established during that initial update. Now, if an argument to your instance changes, only those members that established a dependency to that argument during that initial update will notice the change and will update accordingly (once they are asked to do so).

Now, this may sound all fine and dandy, but it also imposes some restrictions. Let's consider a Feature with the following arguments and Feature-code that creates a circle either using the *FCircle* type or as an *FNurbsCurve* (because a circle can be represented in different ways):

// arguments:

```
FBool createAsCircle
... // more arguments that are needed for either type of circle creation, e.g. normal, radius, etc.
```

//code:

```
if (createAsCircle)
  circle c1(normal, radius, 0, 360)
  FCurve theCircle(c1)
else
  . . . // calculate everything that is needed for the nurbs curve
  nurbscurve c2()
  c2.setEllipse(normal, center, p1, 1)
  FCurve theCircle(c2)
endif
```

This Feature will create a circle in two different ways, one using the “normal” *FCircle* creation, one using the creation method from three given points as an *FNurbsCurve*.

I have omitted the calculations that are necessary for the NurbsCurve creation; if you are interested in the details, just take a look at the “CurveCircleFrom3Points” Feature that is shipped with CAESES.

In the end, the resulting curve from the selected creation method will be accessible through the same member “theCircle” that is either a reference to the *FCircle* or to the *FNurbsCurve* (see Item 5 for more information on references).

Now, if this Feature is set to not recreate on update, consider the initial update with the argument `createAsCircle` set to true, the initial update will create the *FCircle* based on the the relevant arguments and is done.

Of course, an *FCircle* may not offer the functionality the user (i.e. you) desires from the curve, so you decide that you actually do want the *FNurbsCurve* instead. So, you go to your Feature instance and change the argument `createAsCircle` to false. Pop-quiz: what effect will this have on the Feature instance?

I'll cut this short and tell you: NONE AT ALL! Remember! All objects have already been created; the code is not executed again. However, even though you'd expect `theCircle` to be a reference to an *FNurbsCurve* now, it’s still an *FCircle* because the implementation logic was not executed again!

This example shows that whenever the Feature-code includes any branching statements (i.e. `if`, `loop`, `while`, `foreach`, `goto`s) the Feature pretty sure does not qualify for transient execution.

CAESES 4.0 comes with a somewhat reliable detection of cases where non-recreating update behavior will most likely yield incorrect behavior of your Feature instances and will issue a warning for those cases (if you disabled recreation), but it's best not to rely on that warning and to think about it yourself. Another case (which I am not sure of whether the warning mechanism works for) is a Feature that writes files. I cannot think of a single case where a Feature that is designed to write files should be set to not recreate on update (same goes for transient execution, but we'll get there in a minute).

Is Drawable

This setting determines (as the name suggests) whether instances of this Feature should be rendered to the 3DView. This only makes sense, if the Feature actually creates objects that can be rendered (curves, surfaces, etc.). Now, it might seem like a minor slip, if you forget to disable this option if your Feature does not create any drawable type, true. However, this setting implies something else as well: whenever a 3DView is shown all objects of your model that are drawable, set to be visible, and are not filtered by the 3DViews filter setting will be updated. In other words (unless you make the Feature instances invisible), setting a Feature to be drawable will pretty much enable an automatic update for its instances if a 3DView is active. Sometimes you will not even notice this (if the Feature code is rather simple); sometimes it may have a major impact on your project's performance. So, in short, always consider whether your Feature actually creates objects that you need to see in the 3DView.

As a side note: sometimes it might be helpful, to enable this option while developing a Feature that calculates some non-drawable values in order to be able to analyze the Feature's behavior, just remember to disable the drawable option, once the Feature development is finished.

Enable persistent / transient

I already mentioned the different execution modes for Features: persistent and transient. The latter merely executes the given command sequence (once) and places all objects that were created in the Feature (and are set to be accessible members) inside the model (it may also place additional objects in the model, that are not set to be accessible but are needed for those objects to work, i.e. the accessible objects depend on those objects - objects like that will be placed in a special scope, though, called auxiliary). Persistent Feature execution creates one object that encapsulates all objects that were created by the Feature. Those objects can only be interacted with through the arguments of the Feature instance.

You can actually achieve the behavior of transient execution by creating an instance and detaching it (right-click and choose detach).

Similar restrictions apply to transient execution as to instance creation with recreate on update disabled. While conditional execution of control paths in you Feature will unlikely to be a problem here, the other restriction I mentioned earlier, which are Features that write files, may lead to trouble. Again, I cannot think of a reasonable use case for transient Feature-execution and file writing (reading, i.e. importing, is a completely different story).

Update only on user request

In general the update mechanism in CAESES depends on the time a value from an object is needed. When the input to an object changes, an object's state changes to "out-of-date". Now, when anything is requested from such an object, the object needs to go through its update routine. For Features this means their create-code is executed (if the Feature is set to recreate on update). The request for updating may be triggered by any other object that tries to access a calculated value and is in the process of updating itself. Such a request can be either triggered explicitly by the user (press the "run" button in the main toolbar of CAESES or implicitly (e.g. because the object in question is supposed to be drawn in the 3DView).

Enabling the "Update only on user request" option explicitly excludes instances of the Feature from the normal update-mechanism. Such an instance will not be updated automatically even if it is out-of-date" and some other object (including the 3DView) triggers an update request. The instance will only update if the user presses the run-button.

Common use-cases for this option are Features that implement optimization algorithms.

Default preview enabled

When executing a Feature or creating an instance from a menu, a dialog pops up that allows to pre-fill the arguments of a Feature (the dialog pops up for transient execution also when right-clicking a Feature and selecting "Execute Definition"). That dialog allows to enable a preview, which is a temporary instance of the Feature which will be drawn to the 3DView (if the Feature is drawable). This offers the possibility to check the output of the given input before creating an instance or executing a Feature. This is a pretty nice to have feature, actually.

There's a little detail though, that needs to be considered for transient execution and for Features that are set to not recreate on update: This preview will already execute the Feature and when confirming the dialog the temporary instance will become permanent (or is detached for transient execution). So, if the initial execution already changes the state of the system or has irreversible results (e.g. import files, delete files), it is highly advisable not to enable the preview. By disabling the default preview option of the Definition, this can be achieved (of course, the final user of the Feature can still break everything by enabling the preview manually, but as a Feature author you did what you can to protect your users from such an undesired behavior).

Default automatic update

Here's an option I want to convince you to steer clear from. In CAESES we are employing a technique that is called "lazy evaluation". This means that (in general) objects are not updated and values are not calculated unless somebody explicitly asks for it. However, you can be sure that values WILL be calculated when they are NEEDED. This technique gives CAESES the possibility to perform the way it does. If everything would be calculated on the fly whenever something in the model changes, it would be close to impossible to work with the system because it would be busy updating the model with stuff you don't even want to know. Enabling "automatic update" on any object totally blows that concept to pieces. An object that is set to automatically update will perform potentially

expensive calculations whenever its input changes. Even if you are not interested in the results of that expensive calculation at all (at least not at that moment - and remember, the values WILL be calculated, once someone asks for them, no matter if the object is set to automatically update or not).

Maybe there are some special cases where automatic update for an object is required or even necessary for it to behave correctly. But those cases are very rare and can be avoided by accurate “dependency modeling”. I honestly assume that you'll never have that case.

So, in short: just ignore this option!

Arguments

This tab allows you to specify the input for your "Feature-program". There needs to be input, because otherwise your Feature will just calculate the same value over and over again, right? At least that's what I'd expect a meaningful program to do. Of course, if you want to operate on random data, for example, there may be valid use-cases where your Feature does not require additional input. Those cases are rare, though, and keep in mind that a Feature instance that does not depend on external input has no way of going out-of-date, hence it will be a "one-shot" execution (except for when you force an update manually through the run-button). To be fair, for transient use, an argument-free Feature may actually make sense.

There's not a lot I can tell you about arguments that shouldn't be self-explained. Arguments have a name (which is used when referencing the argument in the Feature's code), a label (which is displayed in the object editor of instances of the Feature) and a category (which is used to group multiple arguments in the editor). It can also have a default value which is used if no value is given for an argument.

There are two things (aside from the advanced options) that may require clarification, though: "allow expression" and "required".

The explanation for the "required" tag is rather short, so I'll start with that. That tag is nothing but and an editor hint. This means, arguments that are marked as being required will get a special mark in the editor, but not setting them will not prevent a Feature from being executed. Same goes, by the way, for standard objects that have input that is marked as being required. It is a mere hint to the user that shows that an argument is supposed to have a valid value, otherwise it is unlikely that an object will behave correctly. The same applies to Feature arguments that were marked as being required.

The "allow expression" option involves some more application logic. Let's start from behind. By marking an argument to not allow expressions, you say that only concrete values or objects should be passed to that argument. In turn this means that the value of that argument should not be a value that is calculated by a command.

This is especially relevant to the type of editor such an argument will have when selecting an instance of the Feature or when trying to execute it transiently (in fact, I am not 100% sure whether this flag is considered at all, when a Feature is used without the graphical user interface. I will look into the code and keep you updated). So, basically, when disabling this option for an argument, that argument will get the editor that is specifically designed for its type. For example, if it's an argument

of type *FBool*, it will get a checkbox instead of a text-edit that allows to put in any expression that has a return value of type *FBool* (or a numerical value for that matter - any numerical value can be, implicitly converted to a boolean value - in “good” C tradition - since every number that evaluates to the number "0" is considered to be the boolean value "false" while every non-zero value is considered to be "true"). While the boolean type is the most likely one to be used for "set as expression" = false, other types offer their own editors, as well. Integer and unsigned will give (editable) spin-boxes, *FFile(In/Out)* offer editors that give you the option to use a (operating system dependent) file dialog to select input (*FFileIn*) or output (*FFileOut*) files.

Disabling this option for any other type will also give you a special editor that allows to only select objects of the given type in a convenient drop-down select, but ever since version 3.0, where a similar functionality was added to "expression editors", there is - in my opinion - no real reason to disable this option. EXCEPT! Wow, I almost forgot this. List:

When your Feature takes an argument of a list type, setting “Allow expression” to false has different semantics. Instead of the editor where you have to type the list using the brackets-syntax, you will be presented with a list editor. Each individual entry of the list can then be supplied as an expression.

Attributes (or Members)

All objects that are created in a Feature’s “Create Function” (i.e. in the Feature’s code) will be added to the Feature’s attributes. The attributes tab gives an overview over those objects. Since not all objects are actually relevant and would just clutter up your object tree, it is possible to specify which objects should be accessible. All non-accessible attributes will only be used internally; most of them will be discarded after the execution.

A Feature instance can also act as a type other than its own type. This type is determined by selecting an attribute to be the “Type Provider”. If, for example, your Feature creates an object of type *FBSplineSurface* and you select that object to be the type provider, you can use an instance of your Feature whenever an object of *FBSplineSurface* (or its base-types) is needed. This is more of a convenience than anything else, since you could have the same behavior by just using the “get”-command for that attribute, but it comes in very handy quite often. Another advantage is, that if your Feature does nothing but encapsulate a “custom” curve type, instances of the Feature will actually be shown with the icon for *FCurve*, if the type provider is set to an *FCurve* type.

Syntax

Now that we’ve covered the general stuff (most of which was probably already known from trainings or correct interpretation of the documentation), let’s finally start with the programming part (which was my original motivation to even start this article).

As already mentioned, the FPL is based on C-like syntax. So, if you are familiar with one of the languages, I mentioned earlier, there should be nothing that surprises you.

In general a command is executed by typing its name, and passing the arguments of the command in parentheses. The "hello world" example for this would be `echo("hello world")`. This will execute the global command "`echo`" which takes a string for its parameter and prints it to the console.

Now, this is a global command. There are also those commands that can be executed on objects. The syntax to do so is through the dot ('.') operator: `myObject.command(arguments)`. This would execute the command "`command`" on the object "`myObject`" with the arguments "`arguments`".

A little realistic example for this would be: `myPoint.setX(3.1)`. Assuming that the object "`myPoint`" is an *F3DPoint* (as the name suggests), this command would change the state of said object by setting its X-coordinate to 3.1.

Now, we know how to call global commands and commands for objects. However, there's a third very important category of commands which I haven't talked about so far and that will be covered in...

Item 1 - Creator Commands (Creators)

I already mentioned objects. But, in order to have objects, objects need to be created. If you want to call a command on a curve, you have to create a curve first. In the graphical user interface of CAESES, that is a rather easy task. Click on the button of the curve you'd like to have and, voilà, it's there. In Features you will have to spell out what you want. Let's see how that is done.

The general syntax is `<createCommandName> <objectName>(<arguments>)`.

I sincerely hope that this way of writing it is not too cryptic for you. Maybe an actual example helps:

```
point myPoint(1, 0, 0)
```

This would create a 3D-point with the coordinates x=1, y=0, z=0. Now, it would be possible to create a line with that point:

```
line myLine(myPoint, myPoint)
```

This would create a line with the start point "`myPoint`" and the end point "`myPoint`". While this is perfectly legal, it certainly does not make a lot sense, since this line would be no line, or, more accurately, would be a line with a length of 0. I never claimed that my examples will make a lot of sense, though. Actually, I have to warn you that most of my code examples will not include actual meaningful functionality. I want to keep your focus on the techniques I'm trying to transport, rather than on some especially clever algorithms. CAESES comes with a set of very nice Features (though I must admit, that not all of them follow all of the advices I advocate here), so if you want to have a look at some actual useful sample code, just browse through those.

One thing that may strike you odd, especially if you have some experience with other programming languages, is, that the create command name is not equal to the type of the object. (In the case of the 3DPoint, the create command name is "`point`", while the type of the resulting object is "*F3DPoint*".) Honestly, I cannot tell you the reasoning behind this, but that's just how it is. In general (although the *F3DPoint* is a bad example for this), the create command name is the type name without the leading "F". However, the type name is important in several other constructs of the FPL, so do not forget it (especially since the expression `F3DPoint myPoint()` is valid but has a completely different meaning than the create command we saw above - more details later...). For the creator part, let's just remember:

- The create command name is not the type name. The creator syntax with the type name is valid and will be accepted, but it will certainly not create a new object of the specified type (again, more on that later).

- The create command allows you (and inside Features forces you) to give names to objects.
- The creator allows you to initialize a new object to an initial state through the argument(s) you pass to the create command.

Item 2 - Feature from Selection

A good way to start a Feature is the “Feature from Selection” feature of CAESES. After setting up a (sub-) model, you can turn it into a *FFeatureDefinition* by selecting all relevant objects you’d like to have included in the Feature and select “Create Feature from Current Selection” from the Features-Menu. This will create a new *FFeatureDefinition* that includes the creation of the objects you had selected along with their values and their dependencies. If some objects the selected objects depend on are not selected, those are added to the new Feature as arguments.

While such a Feature does not offer any algorithms, it is usually a good starting point for implementing more complex Features and to encapsulate commonly used structures within a re-usable object (the *FFeatureDefinition*).

Things to remember:

- The “Feature from Selection” option allows you to quickly encapsulate existing (sub-)models into an *FFeatureDefinition* that can serve as a good starting point for a more complex Feature.

Interlude – Fast Forward

Ok, so, I have to apologize, after all my initial idea was to write "Effective FPL". I must admit, I got a little side-tracked and this has turned into a beginner's guide. If I keep going like this, it will really end up to be a book, plus, from what I know, everything up to here should have been covered in your training already (although... there are so many CAESES free users out there that didn't have a training, so I hope some of you have actually learned something up to here).

I am afraid, that I will lose some of my readers after this point, but I hope that I showed you that learning Features is worthwhile since it allows to extend the already manifold capabilities of CAESES.

If you bail out now, just remember that Features allow you to adjust CAESES' functionality to suit close to ALL of your needs! And maybe you come back, once you’ve reached a point where a Feature-less CAESES just doesn’t cut it for you anymore.

So, let's actually dive into some advanced and less obvious stuff here! After this point, I assume that you have at least a basic idea of Feature programming, so I will not go into detail on the very basic stuff, but continue to explain things that I fear to be unclear even for somewhat experienced Feature-artists!

Item 3 - Use Basictypes

Some things in CAESES can be represented by several types. A double (double precision floating point – one of the most fundamental types of all) value, for example, can be held by a variable of type *FDouble*. But, there's another (higher level) type that can hold the same value: *FParameter*.

The difference between a double and a parameter is that a double is a discreet value, while a parameter can hold an expression that can be evaluated to a double value. Both are valid and

important to have, but they do serve different purposes. Using a parameter to store a double value allows to keep a dependency to another object. So, let's say, for example, the number you want to refer to depends on the coordinate of a point, which is calculated depending on the shape of a curve that may change over time, the *FParameter* will keep track of those changes, the *FDouble* will not. So an *FParameter* is a lot more powerful than a primitive *FDouble*.

But, this power comes at a cost. If you really just want to store a floating point number, an object of type *FDouble* is a lot cheaper than using the *FParameter* type. This may also apply if the value does actually depend on another object, but you know that between creating the variable and using the actual value, the supplying objects do not change. The same applies when using *FParameter* instead of *FInteger* or *FUnsigned*. There are other pairs of expensive/cheap (almost) interchangeable objects in CAESES:

FStringParameter / *FString*, *F3DPoint* / *FVector3*.

Things to remember:

- Use objects of basictypes whenever possible as long as you do not want to keep track of dependencies, because they offer better performance.

Item 4 - Temporary Variables

Let's consider the following code:

```
objectlist list()
// fill list with some values
unsigned i(0)
while (i < list.getSize())
  // do something with each element of the list and increase the index
  // variable "i" to avoid an endless loop
endwhile
```

Can you spot anything in this code, you would do differently? If your answer is that iterating a list can be done a lot more efficient using the `foreach` statement you do get credit, but that is not this item's topic, I will talk about `foreach` later. The problem here is efficiency on a lower level. For every iteration here, the size of the objectlist is checked. While this is a rather cheap operation, the cheapest operations are those that are not executed. So instead of calling the `getSize` command of the list for every iteration, it would be a lot cheaper to call it only once before the loop and store its value in a temporary variable (yes, we currently have no optimizing compiler that could translate the code above to the following and I doubt we ever will):

```
unsigned i(0)
unsigned n(list.getSize())
while (i < n)
```

Now, instead of calling `getSize()` n times during the loop, you only have one call and n fetches to a variable, which is pretty much free.

Again, you may say that `getSize()` is constant in time anyways (if you happen to know that fact) and I am talking about micro-optimization here. I must admit, that you are not entirely wrong in this case. But consider the following code that reads coordinates from a file and creates points from those coordinates (I'll omit the file reading, I hope my comments will make clear what is going on):

```
objectlist values()
... // Read lines from a file and put them into the objectlist.
```

```
    // Assume each line that is read contains point coordinates x y z separated by one
    // or more whitespace.
unsigned i(0)
entitygroup points() // an entitygroup that will hold the resulting points
while (i < values.getSize())
  double x(values.at(i).castTo(FString).splitByRegExp("\s+").at(0).castto(FString).toDouble())
  double y(values.at(i).castTo(FString).splitByRegExp("\s+").at(1).castto(FString).toDouble())
  double z(values.at(i).castTo(FString).splitByRegExp("\s+").at(2).castto(FString).toDouble())
  point p(x, y, z)
  points.add(p)
endwhile
```

This code (in a little altered form) is taken from a real-world Feature, I've seen, by the way. Let's take a closer look. Ok, I already told you that the loop checking could be done more efficient by using a temporary variable for the objectlist's size. The second, more dramatic, potential for improving this code is the call to `splitByRegExp` three times on the same string object. Since our lines aren't very long (only three numbers), this is not a huge overhead, but splitting a string, especially when using a regular expression, is definitely something you do not want to make more often than necessary.

In contrast, look at this code that makes use of temporaries:

```
objectlist values()
... // read lines from a file, for example and put them into the objectlist
    // assume each line that is read contains point coordinates x y z separated by one
    // or more whitespace
unsigned i(0)
entitygroup points() // an entitygroup that will hold the resulting points
while (i < values.getSize())
  objectlist tokens(values.at(i).castTo(FString).splitByRegExp("\s+"))
  double x(tokens.at(0).castto(FString).toDouble())
  double y(tokens.at(1).castto(FString).toDouble())
  double z(tokens.at(2).castto(FString).toDouble())
  point p(x, y, z)
  points.add(p)
endwhile
```

I am rather sure that it should be pretty clear that there is no way that the first version could out-perform the second one. This effect gets larger, the more complex the algorithm becomes.

This is important:

- The usage of temporary variables to store return values of commands yields better performance than repeatedly calling the command.

Item 5 - References and basictypes

Ok, so you do know now, that temporary variables are your friend. But what, you may say, if my list contains objects that would be very expensive to copy into a temporary variable? And also, I want to iterate through the list and perform operations directly on those objects that are stored in the list and not on temporary copies of those objects!

Of course, these are valid arguments against copies. But, CAESES offers a mechanism to do just that without having the copy overhead and without having to resort to list accesses every time you want to change an object in a tight loop: references.

Remember when I said that writing a creator with the type name instead of the creator name does something and creates valid code, but it surely does not create a new object? Well that's exactly the syntax we need right now:

```
line l() // create a line with the name "l"
```

```
...      // initialize it and do other work with l
FLine ref(l) // this does not create new line that is a copy of l, it creates a reference to the
             // original FLine object!
echo(“” + ref.getStart().getX()) //prints the x-coordinate of the starting point of l to the console
ref.setStart([8, 7, 6])          // sets the starting point of l to [8, 7, 6]
echo(“” + l.getStart().getY())   // prints 7
```

References are especially useful when accessing an object that's stored inside an objectlist:

```
FLine ref(list.at(5).castTo(FLine)) // create a reference to the sixth object that is stored in “list“
// which should be a line (for type safety, i.e. what happens if
// that object is not of type FLine, see below).
// use “ref“ to play around with the object that's stored in the list without having
// to access the list and type that whole casting stuff over and over.
```

Now, that's pretty convenient, isn't it?

Basically, you can think of a reference to be a new name that you give to an existing object, which you can then use to “talk“ to that object.

Let's look back at that line involving the `castTo` command. As already mentioned, all objects have a type. Types may be related. For example, we have the type *FCurve*. This is the base class for all curves. There are many types of curves, but they all have something in common. That common stuff is defined in the *FCurve* type. So, some operations exist, that can be performed with any type of curve (like asking for their start and end position) while others require special types of curves.

Now, all objects in CAESES share a common type. That type is called *FObject*. While it is an important type internally, it does not offer a lot of functionality for you. However, since all objects share that common type, it is possible to store objects of arbitrary type inside of objectlists. But, that is all the objectlist knows about its objects: they are *FObject*s. It doesn't know whether it is a curve, surface, or only a double value.

So, when using objects from an objectlist, you have to tell CAESES the type of that object, and that is done through the `castTo` command. When that command is executed, CAESES looks whether the given object's type corresponds to the type given to the command, and if there is a match, it will happily treat the object as an object of that type. If it isn't, though, the cast will return a null object. Nothing can be done with the null object, except asking it, if it is null by using the global `not`-command (short: `!`). So to see if a cast succeeded, you can do the following:

```
FCurve c(objectlist.at(i).castTo(FCurve))
if (!c)
  echo("No curve at i!")
else
  ... // use the curve
endif
```

So please remember, that the `castTo` command does not change the type of an object, it just checks the type to be the desired type and if it isn’t it will return NULL.

Now that we've covered casts, let's back to references. I said that references give a new name to an object but do not create new objects. This is true, but it is only almost always true. There is a special breed of types where this does not apply, the basictypes (see also Item 3). Basictypes contain the numerical types (double, unsigned, and integer), strings, bools, and some others that you will hardly ever be confronted with directly. Basictypes are special because it is impossible to create a reference to a basictype. Hence, for basictypes writing the creator command using the typename does, indeed, create a new object of that type (again, the `foreach`-statement behaves a little different here, but that

is one of the many nice things about it - I'll cover it in depth later). The reason why this is forbidden is that otherwise code like this would be valid:

```
FInteger dblRef(9) // create a reference to the integer "9"
dblRef += 1        // what exactly should happen here if dblRef is a real reference?
```

The second line says that the value of the object which the reference is another name for should be increased by one. But `dblRef` does not refer to an object! It refers to a constant expression. There is no way to give this a meaningful, predictable, correct behavior. Just for completeness: the code above creates a "normal" variable (not a reference) `dblRef` of type double with the value 9 and it will have the value 10 after both lines were executed. Creating a reference at that point would mean that every subsequential use of the primitive value "9" would yield the numerical value 10 instead. I do not think that would be expected behavior.

So, when you are iterating a list of basictypes and want to change the values that are actually stored in the list, you will have to resort to accessing the element each time using the `at` command and go through the excessive typing of spelling out the cast command. I will get back to this topic after Item 8, i.e. once we've talked about the different types of loops.

Things to remember:

- References give a new name to an existing object which can be used to access an object
- References are created using the creator syntax and replacing the creator-command's name with the type name
- It is not possible to create references to objects of basictypes

Item 6 - Use base class or basictype arguments

This item goes somewhat hand-in-hand with my advice of item 3 to use basictypes for variables whenever possible. Yes, they do have the drawback of not being able to create references, which makes them less convenient inside lists, but they do have a large performance edge over their non-basic counterparts. What is true about variables is of course true for Feature-arguments, as well.

The "expensive" counterparts to basictypes can be implicitly converted into the corresponding basictype. So even when specifying an argument to use *FDouble*, it is still possible to pass an object of type *FParameter*. On the other hand, when specifying *FParameter* as the argument type, it is not possible to pass a command that returns a double for that argument (e.g. `F3DPoint.getX()`). So, by using the basictype for an argument, you actually open up your Feature to a broader range of uses, because every type that can be converted to the given basictype can be used. Implicit conversions to basictypes are available for many types in CAESES, while implicit conversions to non-basictypes are pretty rare (I cannot think of any right now, to be honest).

Basictypes are one thing, but remember when I talked about the inheritance hierarchy of types? All curves, surfaces - even Features themselves, have something in common. This is done through a common base-class. Objects of a common base-class can be interchanged whenever the required type is the base-class. By applying that knowledge within your Feature for the arguments it is possible to write Features that are applicable to a broad range of objects. If all you need from the curves that are put into your Features are the common properties offered by the *FCurve* class, for example, but you use *FBSplineCurve* as the argument type, you'd limit the possibilities for reuse of

your Feature to B-Splines only, while it would (well, let's say could) work very well for any other curve type.

Now, you may say, the *FCurve* type does not offer much functionality. That's right, but the class hierarchy of objects is much more diverse. Just take a look at a type's documentation and you'll see the full hierarchy of the type you are intending to use. Maybe one of the classes in between the very low-level base-class and your class could be a good choice.

What to keep in memory:

- Using basictypes or base classes for argument types allows to use Features in a broader range of applications.

Item 7 - `loop` and `while`

There are multiple ways to create looping structures which are usually the key to implement even the most basic algorithms. The two most versatile ones are the `loop` and the `while` statements. Their cousin `foreach` is only applicable for iterating through containers (objectlists, entitygroups and their descendants).

The `while` statement does not come with any real surprises, so the only thing I can say is, that you should always make sure to have your while-condition actually terminate to avoid endless loops. CAESES does offer a safety net and will ask you whether you want to continue a loop, once a certain number of iterations have been executed, but especially for complex while-bodies that may take a while. A safe bet is to run your Feature at least once inside the debugger (see the epilogue for more info what that is), where you can always pause execution once you have the feeling that you are running in an endless loop.

Actually, the `loop`-statement is even more fool-proof in that regard, since there is no way to create an endless loop using it. It is pre-defined how many times a `loop` will be executed. One thing you must note here, though, is that the command you give to the `loop`-statement is only executed once, before the loop-body is entered! So you cannot create dynamic loops:

```
objectlist values()
... // initialize list with 5 values
loop (values.getSize())
  values.add(6)
endloop
```

The given loop will execute exactly five times, although by the end of the loop, the list contains 10 items (and `values.getSize()` returns 10).

One lesser known feature of the `loop` statement is the iterator variable `$$i`. That somewhat cryptic syntax will give you the current loop count, so:

```
loop (5)
  echo("" + $$i)
endloop
```

will print the numbers 0 through 4.

Now, it is not possible to create endless loops with the loop statement. However, when using it to go through a container there is still room for error:

```
objectlist list()
loop (10)
  list.add($$i) // fill list with values 0 through 9
endloop

loop (list.getsize())
  if (modulo($$i, 2) == 0)
    list.eraseAt($$i) // erase every second element of the list
  endif
endloop

echo("list size: " + list.getSize())
loop(list.getSize())
  echo("" + list.at($$i).castTo(FUnsigned)) // print the content of the list
endloop
```

Do you spot the bug? The goal is to erase every second element of the list. So, we'd expect to have a list size of five at the end. However, the list size is six and the complete output of the code above is:

```
list size: 6
1
2
4
5
7
8
```

So, we did not erase every second element, but we skipped some and erased the wrong ones, and we ended up erasing every third element (0, 3, 6, and 9). In fact, the last three runs of the second `loop` already had a value of `$$i` that was larger than the list's size at that point. So, if you want to manipulate a list that you are going through in a loop, the `loop`-statement is most likely not the best choice.

Two commands that are offered by all loop types and are rather recent additions to the FPL are `leaveloop()` and `continueloop()`. The former will allow you to step out of a loop at any time and continue execution after the respective end-statement for the current loop. The latter will stop execution of the current loop-body and continue execution with the next iteration of the loop.

- The `while` statement defines loops that keep going until a condition is met
- The `while` statement risks to define infinite loops
- The `loop` statement defines loops that iterate a pre-defined number of times
- Any loop type offers the possibility to use the `leaveloop()` and the `continueloop()` commands

Item 8 - `foreach`

I already mentioned the `foreach`-statement a couple of times. It hasn't been around for all that long, so it might be that its existence has slipped by for you.

The `foreach` statement offers a convenient and type-safe way of iterating through lists of objects. This includes the types *FObjectList*, *FEntityGroup* (and its descendants *FOffsetGroup*, *FOffsetGroupAssembly*, *FPanelMeshGroup*, *FStreamLineGroup*, and *FSurfaceGroup*).

Using the `foreach` statement allows to go through a list without having to worry about what objects may be in there if you are only interested in a certain class of objects. Let's look at an example:

```
objectlist a()
a.add(point())
a.add(6)
a.add(point()
a.add(7)
```

```
foreach (FDouble d in a)
  echo("" + d)
endfor
```

This will print only the numbers 6 and 7. The entries of a type that is different from *FDouble* (or, to be more precise, cannot be converted into an object of type *FDouble*) are skipped.

By default, the "iterator object" (in the example above the double with the name d) is a reference. So any changes done to it are also stored in the list that is iterated. I have to say "in general", because if the type that is found in the list is not exactly the one that is specified in the `foreach`-statement, but the object can be implicitly converted to that type, this conversion will take place. In that case, the iterator object will actually be a copy of the original object that is converted to the desired type. So, if you want to make sure that you always get references for your objects inside the list, you have to ensure that your list contains the desired types. I have to apologize for this inconvenience, but that's just how it is. We've always wanted to offer type-safe containers (i.e. lists that can store only a specified type of objects) to take that burden from you, the user, but so far that feature is not available.

Unlike the `loop`-statement, there is no way to find out the index of the current iterator object in the list. This is mainly due internal implementation details for efficiency and also due to the fact that some objects in the list will be skipped. So, unless you are sure that all of the objects in your list are of the iterator type, even counting yourself will not suffice to find the index of a certain element. If you do need the index, you are better off, using one of the other two loop-types.

One word of warning: Unlike the `loop`-statement, the `foreach`-statement is not blind to changes that are done to the list that is iterated through. So the following code:

```
objectlist a()
a.add(1)
foreach (FDouble d in a)
  a.add(2)
endfor
```

will actually create an endless loop. So just remember to steer clear of changing a container that is currently iterated over using `foreach`. Those cases may not always be as easy to spot as above. Even code that may seem guarded against the problem may contain it:

```
objectlist a()
a.add(2)
a.add(6)
a.add(3)
a.add(9)
foreach (FDouble d in a)
  if (d == 6)
    int idx(a.indexOf(d))
    a.insertAt(idx, 5)
  endif
endfor
```

Now, it's not as clear as above, but this will also create an endless loop. Once the value 6 is reached in the `foreach`, the code identifies the index of the current double-object and inserts a new object at that position. What does this mean for the `foreach`-statement, though? For the next iteration, the iterator is advanced one position and if the object at that position is of type *FDouble*, the `foreach`-body is executed. Let's dissect this further, during the iteration where the 6 was found, the iterator position was 1. The body of the `foreach` then inserted the value 5 at position 1. So in the next

iteration, the iterator will be at position 2. What does it find there? Exactly! The 6 again! The same applies for every following iteration, of course. So again: Do not change the container while running through it using `foreach`. To be fair, conditionally adding objects to the end of the container, may be fine depending on the condition.

I want to add something else regarding the syntax: Actually, there are multiple valid definitions of a `foreach` statement. The following are all valid and define the same `foreach`-loop:

```
foreach (FDouble d in a) //declaration of foreach using C# like syntax
  // …
endfor

foreach (FDouble d : a) //declaration of foreach using almost C++ like syntax
  // …
endfor

for (FDouble d : a) //declaration of foreach using real C++ like syntax (valid since CAESES 4.0.3)
  // …
endfor

for (FDouble d in a) // declaration using a mixture of the above
  // …
endfor
```

In other words: the keywords "`for`" and "`foreach`" are equivalent and the "`in`"-part of the foreach-declaration can be replaced by ":". I just wanted to mention this, because I tend to use the ":" instead of the "`in`" out of C++ habit and because it is shorter to type and I don't want to throw you off and keep you puzzling over it when I show you code.

Things to remember:

- The `foreach` statement is a nice tool to implement type-safe iterations of containers.
- The `foreach` statement will always create references to the objects contained in the container, even if they are basictype objects
- Do not change the container inside the `foreach`-body

Interlude – Back to references and basictypes

Ok, now that we know all there is to know about the different loops, let's revisit the topic of references and basictypes. Consider the following code:

```
// define a convenience function that prints the content of an objectlist, see Item 10 for functions
function printList(FObjectlist l, FString op)
  echo("------ The list content " + op + " is ------")
  foreach (FDouble d : l)
    echo("" + d)
  endfor
endfunction

// create an objectlist and fill it with some values
objectlist l()
l.add(1)
l.add(2)
l.add(3)

printList(l, "at the start")

// go through the list and assign a new value to each element
unsigned i(0)
unsigned n(l.getSize())
while (i < n)
  l.at(i).castTo(FDouble) = i
  i += 1
```

```
endwhile

printList(l, "after accessing each element directly through .at().castTo()")

// go through the list and try to access each element through a reference
unsigned i(0)
loop (l.getSize())
  FDouble d(l.at($$i).castTo(FDouble))
  d = d*2     // This does not do what you may think!!!
  i += 1
endloop

printList(l, "after using a \"reference\"")

// go through the list using the foreach statement
foreach (FDouble d : l)
  d *= 2
endfor

printList(l, "after using foreach")
```

What output would you expect here? Well, I already told you that it's impossible to create a reference to a basictype (which *FDouble* is), so I hope your idea was to say that the output will start with

```
------ The list content at the start is ------
1
2
3
------ The list content after accessing each element directly through .at().castTo() is ------
0
1
2
------ The list content after using a "reference" is ------
0
1
2
```

Great! If that was not your idea, let me stress again: It is not possible to create a reference to an object of a basictype. The reason for this is explained in item 5, so go back there if you have the feeling that it should be possible.

Now, what output will be produced after the `foreach`-statement has finished? Let's take a look:

```
------ The list content after using foreach is ------
0
2
4
```

Wow! The list content has actually changed! In other words, the `foreach`-statement does indeed always create a reference to the objects in the list that is iterated through, even if they are of basictypes. This doesn't clash with the reasoning behind not being able to create references to basictype objects. The list will never contain a primitive value; it will always contain actual objects of the basictype.

Item 9 - Nested Features

Features are not necessarily "alone". Of course, you'll say, they may use a bunch of objects from your model that are passed to the Feature as arguments. However, I am thinking of something else here. Features may use other Features to do their work. Those Features (to make this clearer – I am talking

of *FFeatureDefinition*s) may reside in the project. Features inside the project can be used without any further work. They are readily available as types and can be created inside any Feature by using the `fp_<name of the FFeatureDefinition>` command:

```
// Assume we have a FFeatureDefinition with the name “myDefinition” in our project
fp_myDefinition featureInstance(<arguments>)
… // use the instance we have just created (i.e. change arguments, access calculated values etc.)
```

Another possibility is the use of “nested Features”. Those are *FFeatureDefinition*s that are part of the surrounding Feature and are not visible outside the definition. This helps to a) not fill up your project on the top-level with Features that serve only a very specific purpose inside another Feature, b) allows to create “self-containing” Features, i.e. Features that are easy to pass around, and c) allows to encapsulate the behavior of your outer Feature to not depend on changes that are done to another Feature in your project.

Besides the fact, that nested Features are not accessible from outside the outer Feature, they can be used exactly the same as a Feature that comes from the project.

Nested Features can offer a convenient way to modularize the implementation of individual Features. Re-occurring tasks can be refactored into a nested Feature stripping the original Feature from the same code over and over. However, there’s another way of achieving pretty much the same thing only with better performance, which brings us to the next item.

Item 9 in a nutshell:

- Nested Features are FFeatureDefinitions that are only visible inside the surrounding Feature
- They can be used to modularize your Features

Item 10 - Functions

Another way to modularize the implementation of Features is the usage of functions. Functions allow to achieve very similar things like nested Features do, but they usually do so in a more efficient manner, because in contrast to nested Features, calling a function inside a Feature does not include the overhead of creating actual objects. Instead the function call is handled directly in the compiled Feature causing hardly any performance overhead compared to inserting the code directly into the Feature whenever a function is called. Especially for recursive algorithms this yields a lot better performance.

We already saw an actual use of a function in the Feature I presented in the interlude after item 8. There we had the convenience function that prints the content of a list of doubles to the console. Let’s look at it again (a little stripped down to its pure functionality):

```
function printList(FObjectList l)
  foreach (FDouble d in l)
    echo(“” +l)
  endfor
endfunction
```

I think this function should be pretty straight-forward to understand. It takes an objectlist as its parameter and prints all doubles in that list to the console. Additionally, functions can have return values:

```
function buildSum(FObjectList l) : FDouble
  double sum(0)
  foreach (FDouble d : l)
    sum += d
  endfor
  return (sum)
endfunction
```

Again, this should be fairly easy to understand: the "`buildSum`" function takes an objectlist and returns the sum of all doubles in that list.

Functions offer close to the same functionality as nested Features do. Of course, nested Features ARE more powerful for some use-cases and may provide more flexibility. For example, it's currently not possible to declare default values for function parameters. Nested Features can also represent objects. So if repeated evaluations are needed on the same input, nested Features may actually yield better performance than repeated function-calls (because the nested Feature instance only needs to be updated once, while the function runs every time you call it, this sounds more like a case for a temporary variable, though - see item 4). If the nested Feature is used as a single-shot functionality, calculating a certain value, for example, or formatting a string, the function will always be faster, sometimes beating the nested Feature by magnitudes.

A little quiz: What do you think, how are objects passed to the function and how are they returned? Will any copies be created or are we working on references to the objects that were passed in?

Times up, but I'll tell you: A function will ALWAYS take it's parameters as references and always give its return value by reference. But how can this be? I said references to basictypes are not possible. Well, for functions the same applies as for the foreach-statement. At the moment a function starts its execution, it will actually work on an object. It always will. Even if that object might be a temporary object that was created from a primitive value that was passed to the function.

To be honest, I am not exactly sure whether we made the right choice to unconditionally pass all arguments to a function by reference. I mean, it is very well possible, that you actually want to pass a number that is stored in an integer variable "`i`", for example, to a function by value, i.e. pass a copy of `i`'s current value, so you are able to modify it without changing `i`'s value at the caller site. Consider this code:

```
function findNextDuplicate(FObjectList list, FUnsigned k) : FInteger
  double val(list.at(k).castTo(FDouble))
  k += 1
  while (k < list.getsize())
    if (val == list.at(k).castto(fdouble))
      return(k)
    endif
    k += 1
  endwhile
  return (-1)
endfunction

objectlist l()
// add double values to l, possibly containing duplicates
unsigned i(0)
while (i < l.getsize())
  unsigned j(findnextduplicate(l, i))
  if (j != -1)
    echo("the next duplicate of location " + i + " is at location " + j)
  else
    echo("no duplicates of the value at " + i + " was found")
  endif
  i += 1
endwhile
```

Now, this looks like very valid and functional code to find the next duplicate for each element of the list. Except that it does not work! It compiles fine and it may find some duplicates, but it will most likely not find everything you were looking for. The problem is that the function modifies the *FUnsigned* "`i`" that was passed to it. Thus, the next iteration of the while loop at the main thread will use the modified value of `i`, which might even be equal to `l.getsize()` if no duplicate was found for the current value. However, that's how it is at the moment, but I'd prefer to come up with an explicit way to pass (basictype arguments) by value instead of by reference.

Another thing, you should consider is scoping of variables. If you never heard that term with regards to programming, don't start to sweat just yet. Scoping considers the life-cycle of a variable, i.e. at what point in time is a name a valid name. Outside of functions, Features do not consider scoping. A variable "lives" forever (well, at least until the Feature has finished executing) after it was declared. Even if the variable was declared in a conditional statement:

```
if (aCondition)
  double d(9)
else
  double d(10)
endif
echo("" + d)
```

Unlike in most other programming languages, this is completely valid code and will print either 9 or 10 to the console (depending on the value of the boolean value of `aCondition`). Now, what would happen if the else part of the if-statement failed to declare the double "`d`"?

```
if (aCondition)
  double d(9)
endif
echo("" + d)
```

The answer may come as a surprise to you, but the code would stay valid and the output would be either 9 or 0! How can that be? This is another specialty of basictypes: they are always initialized, i.e. an object of a basictype can never be undefined (NULL). So even though the object "`d`" never ran its creator if "`aCondition`" is false, it is already initialized to the double's default value (which happens to be 0) even before any code is executed.

Now, what happens if we do not have a basictype object?

```
if (aCondition)
  point d(9, 9, 9)
endif
echo("" + d.getX())
```

This will either print 9 or the echo line will do nothing at all, because it will yield an invalid command (the `getX()` command is called on an undefined object). I hope this makes clear that while conditional construction is valid, it might lead to subtle bugs if you are not careful.

Ok, I got a little side-tracked here, we were talking about scoping. I think the above shows that outside of functions, you will never have to worry about scoping. A variable name is valid starting from the point where it was defined.

Now, when functions enter the ballpark, things change a little. A function can access objects that were defined outside of the function but not objects that were defined inside another function. Only

the function itself can access an object that was defined inside of it. Defining an object inside a function that has the same name as an object that was defined outside of the function will hide the outside object and that object will no longer be available to the function; this includes the parameters of the function. I think this is easier to understand if we look at some code:

```
function func1()
  double d(1) // local variable, hides the global variable with the same "d"
  echo("" + d) // prints "1" (the value of the local variable)
endfunction

function func2()
  d = 2 // changes global variable
endfunction

function func3(F3DPoint d)
  d.setX(3) // change the object passed as an argument
endfunction

function func4()
  double g(4) // local variable
endfunction

function func5() : FDouble
  double g(5) // local variable
  return (g) // which is passed back to the caller
endfunction

double d(0)
func1()
echo("" + d) // prints 0 - func1 only changes a local variable
func2()
echo("" + d) // prints 2 - func2 changes the value of the global variable "d"
func3(d) // will not compile - type mismatch between FDouble and F3DPoint
point p()
func3(p)
echo("" + p.getX()) // prints 3 - func3 changed the x-coordinate of the point that was passed to it
func4()
echo("" + g) // will not compile - g was declared inside of func4 and is not available from the outside
d = func5() // assigns the return value of func5 to the global variable "d", the name "g" that was
// used inside of func5 is no longer valid
echo("" + d) // prints 5
```

I think you got the point, right?

Basically, functions allow to specify global commands that are valid inside a Feature. But, they do not only give you the possibility to extract re-occurring parts of your Feature into a sole place of code, they also allow you to alter the behavior of built-in global commands inside the feature, because they can hide existing global commands if they have the same signature:

```
function echo(FString str)
  // This function hides the global echo command
  // but we can still access the existing global command by preceding its name with "::"
  ::echo("My Feature says: " + str)
endfunction
```

By declaring that function, whenever the "`echo`" command is used in that Feature, it will actually call the function instead of the global command, which in turn calls the global command `echo`.

Things to remember:

- Functions are a technique to modularize the implementation of complex Features
- Functions can alter the behavior of built-in global commands by hiding them, the global command is then available by prefixing it with "::"
- Functions always take their parameters by reference

- Functions follow scoping rules

Item 11 - Nested Features revisited: Custom commands

Functions usually offer a better performance when used for modularizing Feature code than nested Features do. However, there is one thing that functions cannot do, but nested Features can: create commands for your Feature that can be executed on instances of the Feature.

Any nested Feature can be called as a command on an instance of the outer Feature. The command has the name of the nested Feature, takes the arguments of the nested Feature, and returns an objectlist with the members of the nested Feature that are set to be accessible. If only one member is accessible, the return value of the command will be exactly that member (without it being nested into an objectlist).

Let's look at an example. Let's assume we have a Feature with the name "BestFeatureEver". In that Feature a nested Feature is defined called "transmogrify" that takes an *F3DPoint* and an *FDouble* as an argument and creates a new *F3DPoint* that is somehow calculated (transmogrified) from the input. That new *F3DPoint* is set to be accessible and is the only accessible attribute of the nested Feature. Let's further assume that we have an instance of the "BestFeatureEver" called "theGreatest". If all of those assumptions are true, the following will be perfectly legal (not only would it be legal, it would also work!):

```
fp_BestFeatureEver theGreatest() // our instance
point p()     // a point
double d(9.7) // a double
F3DPoint newPoint(theGreatest.transmogrify(p, d)) // create a new point using the nested Feature
                                                  // and store a reference to it
point anotherPoint(theGreatest.transmogrify(newPoint, 42)) // create a copy of the point that was
                                                           // created by the nested Feature
```

This technique offers a way to create "full" custom types, i.e. types that do not only hold custom data and have custom behavior, but also offer custom commands.

Now, we know that commands of objects usually work on that object's data. Calling `setX(5)` on an instance of an *F3DPoint* will change the data of exactly that *F3DPoint*. For commands of Feature-instances that are implemented through nested Features, however, this is not exactly the same. The nested Feature itself is a different type than the outer Feature (in the example above one is of type *FFeature::BestFeatureEver*, while the other one is of type *FFeature::transmogrify*. So when the *transmogrify* Feature is executed, it does not know anything about the instance of the *BestFeatureEver* you are actually invoking the command on. There are two ways to achieve a behavior that is somewhat similar to the behavior of built-in types and allows you to work with the data of the outer Feature:

1. Pass the surrounding Feature-type as the first argument of the inner Feature. This is actually the most common approach. It will, however, change the command call to look somewhat "unnatural". For example, if the *transmogrify* Feature would need to work on the data of *BestFeatureEver*, the call would look like this: `theGreatest.transmogrify(theGreatest, p, d)`
2. Use the `getParent()` command inside the nested Feature. That command returns the type *FFeature*, so you have to resort to a cast when using it. For our example, the *transmogrify* Feature would then include the line: `FFeature::BestFeatureEver outer(getParent().castTo(FFeature::BestFeatureEver))`, which creates a reference (see item 5)

to the instance of the outer Feature. This approach is some typing overhead for the nested Feature, but the resulting command call will feel more natural.

What to remember:

- Nested Features allow to implement custom commands for your Feature-type that are callable from the outside.
- In order to implement “real member functions” (i.e. commands that work on the instance of the outer Feature), either pass it as an argument to the nested Feature or use the `getParent()` command.

Item 12 - Type out default arguments if possible

Parameters of commands sometimes have default values. Those values are taken if no argument is supplied. The compiler resolves arguments from front to back, so in order to be able find out, which parameters should be filled with default values, all parameters that follow a parameter with a default value need to be defaulted as well (otherwise CAESES would not know which parameter a supplied argument belongs to).

In turn this means when a command has, for example, three parameters and all of them have default values, but you only want to specify the second one, you will have to type out the first argument, too (and you'd want to use the default value there). Defaulted parameters are convenient, because they a) save you some typing and b) give a good indication of what arguments are actually required to make a command work and which ones are optional. So far, this is not specific to Features, it applies to normal expression editors as well; I just wanted to lay common ground, so you all know what we are talking about.

When using commands in a Feature (which is basically all a Feature consists of – at least the Feature code) this convenience can be used, too, of course. However, inside Features it comes with a cost. Before I go into the gory details of why that's the case let me give you the "too long didn't read" (tl;dr;) version: using default values for command-parameters in a Feature comes at a runtime cost. It's not a huge overhead, but when using them often (e.g. in a loop or on many commands) it is definitely measurable. So, instead of relying on the default-value mechanism, find out what those values are and type them into your command call:

```
point p()        // uses default args
point p(0, 0, 0) // same as above with default args spelled out
```

In order to find out what those default values are, either look into the type's documentation or move the cursor into the parentheses behind the command and press the autocomplete short-cut (CTRL-Space). The auto-completion shows the complete signature (i.e. declaration) of the command. All parameter names that have a default value are followed by an equal sign and the default value. The auto-completion for the point creator above, for example, would look like this:

```
F3DPoint point(FDouble x=0.0, FDouble y=0.0, FDouble z=0.0)     [ … some documentation … ]
```

This means the command "point" returns an object of type F3DPoint and takes three FDouble objects as parameter and they are all defaulted to the value 0.0. This is the fastest way to figure out the default values for the parameters of a command. When using a command, check if there are any parameters with default arguments, and if there are, type them into the command call!

Now, this was the tl;dr; version, but I promised you some gory details. After all you may not only be reading this to write more efficient Features, but also to better understand what is actually going on under the hood of CAESES. So let's enter the dark caves…

What happens, when you compile (i.e. press apply) in a Feature that is using the first command call, the one that is using the default values? The compiler will try to find a creator command (because it is using the creator syntax) "point" that takes no arguments. It succeeds to do so, because it finds the command where all parameters have default values. But the compiler doesn't care what those values are. It just knows that the command can be called without arguments, the rest has to be taken care of by the command itself (i.e. fill the missing arguments with its default values). This "taking care"-part will be done at run-time. So, when the Feature is executed, the command will fire up the compiler again to find out the default values for the arguments that are missing. In my example this would not be very costly, because it isn't that hard to find out that "0.0" evaluates to the double value 0.0, but it's still unnecessary. Also, some other default values may not be as easy to compile. A defaulted FVector3 argument with the value “[0, 0, 0]” may not look a lot more complex than the double example above, but it does involve a lot more: the string to compile is a lot longer (actually three times longer, spaces do count – but they are cheap, I have to admit) and the value that is compiled to be an objectlist which is then implicitly converted to an FVector3. So it involves a longer string to read in and involves some more internal magic to make this whole thing work.

If I would have written out the second version right away, those (constant) values would have been evaluated at compile-time of the Feature and placed into its "constants section” of the compiled code. Getting something from there is (basically) free. Now, before you think too much about that “constants section”: I promised gory details, but I won't go into that so far to even try to explain that part to you (in order to keep the PG13 rating), so just accept that there is a "constants section" and that the compiler does do some fancy stuff.

The reason why the compiler cannot do this itself is that default values are not necessarily constant. This is especially true for Feature-creators (I.e. fp_...) or Feature executors (ft_...) which may take any valid object as a default value. That special class of commands is not the only case where this applies, though.

Things to remember:

- Spell out default values for commands. The documentation and the auto-completion help to find out the actual values.

Item 13 – Use `switch` instead of cascading `if`s

You may know the switch statement from other programming languages. It is a conditional construct that enables multiple paths of execution based on a single value. In this regard it is very similar to the normal if-statement. However, the if-statement only knows two paths: the "condition is true" path and the "else" path. When your algorithm needs to react on a single value in more than two ways, using the if-statement this would require an if-elseif-else cascade. This works, but when the conditional value is of a certain type (namely one of the integral type or a string) the switch statement is much more efficient. Consider the following (stupid – please send some good examples) code:

```
unsigned var(8)
```

```
if (var == 9)
  echo("var is 9")
elseif (OR(var == 7, var == 10))
  echo("var is 7 or 10")
elseif (var == 0)
  echo("var is 0")
elseif (var == 8)
  echo("as expected, var is 8!")
else
  echo("var is none of the above")
endif
```

Using the switch statement the (semantically) same code would look like this:

```
unsigned var(8)
switch (var)
case 9
  echo("var is 9")
case 7, 10
  echo("var is 7 or 10")
case 0
  echo("var is 0")
case 8
  echo("as expected, var is 8")
default
  echo("var is none of the above")
endswitch
```

On first look, it may seem like the switch-version just saves some key-strokes. However, that's not the whole story (and it's not that many keystrokes, anyways… 203 vs. 240 characters). The `if-elseif-else` version evaluates the conditional command that compares `var` with a given value five times in the worst case (i.e. the case when either the last `elseif` or the `else` branch is executed). The switch statement only evaluates the value of `var` once and will jump directly into the branch that applies. This is possible because the `switch`-statement builds up a lookup-table that allows it to jump directly to the branch that needs to be executed for the value found. So, basically, you can think of it as if the switch statement automagically knows that, when it encounters the value "0" in line 2, it can directly continue with line 8 and go on with the next line after the `endswitch` statement afterwards (of course, this is not really how it works, all the compiler sees is the so-called bytecode and not any lines of code, but it's to illustrate the point here.).

What to remember:

- When an algorithm has more than two paths of execution based on an integral value (*integers*, *unsigned integers*, *strings*, or *enum* values), the `switch` statement is more efficient than cascading `if-elseif-else` statements.

Item 14 – Creating multiple objects in a loop

I have to admit that this last item is a little different from the other ones. While the previous items cover general techniques regarding Feature-programming, this one covers a very special use-case. However, this is a reoccurring question and problem for many people and it is also very commonly needed in Features. That's why I decided to include it here (besides, there's no way I could stop with number 13! I mean, I'm not superstitious or anything, but still…).

A pretty common scenario for algorithms, especially those that are useful in a geometry-focused system like CAESES, is creating multiple objects within a loop, for example to create points along a certain path that is calculated in a loop:

```
unsigned i(0)
```

```
while (i < 10)
  point p(i, 0, 0)
  i += 1
endwhile
```

This is supposed to create ten points in a loop along a straight line. The problem is, it does not work. All points are stored in the variable `p`. Now, this has the effect that the value of `p` from the previous iteration is overwritten each time and in the end we will be left with exactly one point – the last one – at `[9, 0, 0]`. Ok, now this is not the desired result. After all, we want to have ten points along a line.

There are two techniques that can be applied to achieve the desired behavior: storing the created objects in an *FEntityGroup* or using so-called "persistent sections". For the code above, both have the same effect, but there are some differences, especially for transient vs. persistent use of Features.

Option 1 – Use a persistent section

Using this technique, the code above would look like this:

```
unsigned i(0)
while (i < 10)
  beginPersistentSection()
  point p(i, 0, 0)
  endPersistentSection()
  i += 1
endwhile
```

For a persistent (drawable) Feature this would achieve the desired result of having ten points displayed in the 3DView in a straight line. However, there is one problem: After the while-loop has finished you can only interact with the last point that was created. The other nine points still exist, but there is no way to access them. This problem can be solved using option 2 (see below).

For transient execution of the Feature this will actually create ten points in your model, which is exactly what we want.

Option 2 – Store the objects in an entitygroup

Using this technique, the code above would look like this:

```
unsigned i(0)
entitygroup l()
while (i < 10)
  point p(i, 0, 0)
  l.add(p)
  i += 1
endwhile
```

This solves the problem of option 1 for persistent Features, since it is possible to now address each point through the entitygroup `l`. Note, that for this to work, though, the entitygroup `l` needs to be an accessible member of the Feature, the accessibility of the point p does not matter.

For transient execution the behavior is actually the same as in option 1. The only difference here is, that additionally to the ten points an object of type *FEntityGroup* will be created in your model as well (assuming that it is set to be accessible). Most likely, this is not what you want.

So in short: For transient execution use a persistent section for this and for persistent use the entitygroup offers more flexibility.

Epilogue: The editor, the debugger, the profiler, and snippets

The Feature editor has improved a lot during the last couple of years and has evolved from a mere text editor with some syntax-highlighting and auto-complete functionality to a very powerful editor. The capabilities of the editor are explained in depth in CAESES' inline documentation. In order to access it, just open the editor, place the text-cursor outside of a word, and press F1 (or click on the little question mark button in the editor's toolbar). This should cover pretty much everything there is to know about the editing capabilities.

Most of it should be known from other IDEs (Integrated Development Environments), if you've ever worked with one. If you do happen to miss a key-function of your favorite IDE, just drop us a note and we'll see what we can do. To be honest, the only reason I even mention the editor here is because I kind of grew attached to it during the time I spent redesigning and re-implementing it from its original basic implementation.

However, there are three functionalities that might be extremely helpful when implementing Features and may still not be known to everyone: the debugger, the profiler, and snippets, which are all part of the editor.

Snippets

Even though it is the last item in listing above, I'll start with this one. The other two help once a Feature is written, this one in helps in actually writing it. Snippets are little pieces of source code that can be inserted into your Feature anywhere you like (either by using the tool-button in the editor or by pressing Ctrl+Alt+V). There are pre-defined snippets for creating all sorts of objects. The advantage of using such a creator snippet over a hand-written creator is that the snippet will also generate (initially commented) code to set all of the object's attributes. This not only may save you a lot of typing, but it may show you some capabilities of the objects you didn't know about, yet. Besides creator-snippets the editor also comes with pre-defined "code templates". Those include the control structures (i.e. if – (elseif) – else, while, loop, foreach, switch, and function). But also there are two special templates for reading and writing files (since 4.0). Using those templates, the boilerplate code that is necessary for opening a file for either writing or reading and doing all the error checking that is involved in that task is readily available. All you'll have to do is provide the file name and do the actual processing.

One recent addition to snippets is the possibility to create your own snippets. The "Code templates" part of snippet menu has an entry "Edit Custom Snippets…" which allows to do just that: create your own reoccurring code templates. I hope it is documented rather well (just click the documentation button), so it should be pretty straight forward to use.

Debugging

The debugger allows stepping through your Feature-code with a pre-defined set of arguments and investigating what your Feature actually does at run-time. After starting the debugging mode (which requires to apply – i.e. compile – your Feature) the arguments can be filled in an editor that is similar to the object editor of actual instances of the Feature. The arguments will be pre-set to the default argument values, you specified on the "Arguments" tab of the editor. Additionally, you will need to tell the debugger at which points of your code the debugger is supposed to enter "break-mode". In break-mode, it is possible to examine the values that were calculated/created so far, set special

values to watch out for, or even execute commands in a special console that can access objects that are part of the Feature.

When using functions, the debugger also gives an overview of the so-called “call-stack” which is basically the trace of functions up to the current point.

Break-points can also be modified to include conditions. This tells the debugger to only enter the break-mode if the condition that is given evaluates to true.

Another way to enter the break-mode is to use the “pause”-button at any time during the Feature execution. This is especially useful when debugging long-running loops.

Profiling

After you achieved the state that your Feature does what you planned on it doing, you may experience the desire to improve its performance. A tool that can help you with that is the profiler. This tool (which is part of the debugger) runs the Feature once using the supplied arguments and measures the execution time for each line. Afterwards, the lines will be colored based on the percentage of time each line took to execute (compared to the total running time). Additional information (besides the color) can be obtained by hovering the line with the mouse which pops up a little window that not only shows the numeric value of the percentage, but also the absolute time and the number of times a line was executed, which might be very relevant for loops.

There are two things to be in kept in mind when using the profiler, though.

- A single run may not create a representative profile for your code. There are many unpredictable conditions (e.g. current CPU load) that cannot be taken into account within the profile. So my advice would be to have multiple runs of the profiler with the same input and evaluate the numbers based on the best results you get.
- Sometimes the work a line does may not be clear right away. This is due to CAESES’ lazy evaluation mechanisms. This means, that a command may not be executed until its results are needed and not necessarily in the line it was typed in.

Closing remarks and a glimpse into the future

Oh boy. For some reason I keep on completely failing when trying to write short and to-the-point articles about stuff that I feel like sharing knowledge about. This has turned into a book instead of an article! I apologize for the length, but I sure hope that you did learn something new today (or during the days it took you to read all of this). And maybe - just maybe - you did find some of this stuff actually interesting and had a little bit of fun. Either way, I hope you follow some of the guidelines and advices I've given, in order to fully exploit the power of CAESES' Feature technology.

I will try and keep this document updated if I (or someone else) notice any mistakes or glitches that have slipped in here. Also, by no means, I consider myself to be the guru of Feature programming. To be honest, although for the last couple of years I took great part in implementing a lot of stuff around Features and the FPL, every once in a while I get my hands on a user-written Feature that keeps me

puzzled for quite a while and - once I eventually understand what's happening - and then I am surprised what's actually possible; not only using Features, but using the capabilities of CAESES in general. It's always great to see how users exploit the capabilities we offer in ways we never thought of. So, I encourage you to contribute any good ideas, advanced techniques, or just pure magic that you employ within your Features. I am aware that many of our customers are not too keen on sharing their success-secrets, but always remember, the more people share their advanced techniques on this matter, the more likely it'll be that even the most advanced Feature-artists among you will find a new tool that they can add to their tool-box for Feature-programming.

All that is now left for me to say is "thanks for reading" - BUT WAIT! I promised a glimpse into the future, you are right.

Ok, let's first take a look at actual current development and afterwards I will say some things about stuff that I would like to implement (or have implemented).

If you read my developer's blog (if you don't know it, just go to our forum, locate any post from me and click the link in my signature - if you are having a hard time finding one of my posts, the Feature-programming forum is a good place to start), you'll already know about this, and I will not go into just as much detail here as I did there (after all, you can just read the blog post). So, here we go: With the current development, some things that are possible right now will no longer work in future releases. Those are things, that we always discouraged people to do, but it worked, so I have to assume there are several people out there that use it. I know I am beating around the bush, so here we go: Very soon **it will no longer be possible to change the arguments of a Feature within the Feature itself**. Think about it: arguments of a Feature are suppliers of the Feature, this means, whenever the argument of a Feature changes, the Feature needs to be updated. For "normal" objects (i.e. non-Feature-objects) it is already impossible (and has been ever since CASES or the FRIENDSHIP-Framework, for that matter, was released) to set an object 'a' to be the supplier for object 'b', if 'b' is already a supplier of 'a' (the same is true if the client-supplier relationship is not as straight-forward, i.e. if there are objects in between). Now, with the ability to change the argument (which, again, is a supplier of the Feature) of a Feature within the Feature's code, you basically create a recursive dependency. The Feature changes the argument (which, to be exact, makes the Feature a supplier for the object passed as an argument), which in turn (should) invalidate the current values of the Feature. Just think about it. Think about it again. Maybe now, it makes sense. If it doesn't, keep on thinking, eventually, you'll understand (I am sure, you will). Now, this doesn't sound right, does it?

Now, we've added a safety net that prevents the Feature to go out of date, if the arguments change while the Feature itself is in the process of being updated. Without that, such a relationship would create an endless loop of updates and set-out-of-dates, which can only be stopped by killing the CAESES process.

However, even with that safety net active, is this really what you want? After all, altering the Feature's arguments within the Feature leaves your model in an undefined state, since the values that are calculated within the Feature may no longer correspond to its argument values. This is surely an undesired state. That's why we got rid of the possibility of you falling into that trap. Don't worry, we did identify one very valid reason to do stuff like this (show values that are calculated inside the Feature directly in the objecteditor), and we came up with a (in my opinion) far superior solution. Please take look at my already mentioned blog for more information.

While the above is all true, we did also come up with a solution for cases where it may be not only desirable to have the old behavior, but also completely valid. The two applications where that is the case are Features that are only executed transiently and Features that are set to update only on user request. In both of these cases the Features (or better its instances) are not really part of the general update mechanism of the model, so for those cases changing the value of the arguments inside the Feature does not leave the model in an inconsistent state. That's why it is allowed to change the arguments of a Feature if that Feature is only set to be executed transiently or if update only on user request is enabled.

So, if I am not completely off, that's it for the foreseeable future. Of course, there's also the future that is only seen by fortune tellers. Let me tell you some things, I envision.

I would like to offer a way to have type-safe containers. Yes, the `foreach`-statement takes away a lot of the burden to write out all those casting, but in my humble opinion it would be a lot nicer to be able to tell the system beforehand that you want a collection of curves, for example, and avoid letting anything else in the first place.

Sorting of containers: in general, you may think, sorting of a collection of values shouldn't be a problem. In general, I say, you are right. In fact, in our forums there you may be able to locate a topic where I posted a project that includes implementations of many different sorting algorithms (I think, the only one of the common ones that I missed is heap sort, but I will leave this as an exercise for the reader [I always wanted to write that!]).

BUT! Remember, containers are not type-safe (see above). For general sorting capabilities, what comparison function should be applied to objects of arbitrary type? So, for a generic sorting command of containers, one pre-requisite would be type safe containers (otherwise the best sorting we could offer is based on the address of the objects in memory and that wouldn't be very helpful, would it?). But even then, it might not be exactly clear what you want a container of points to be sorted by. The x-coordinate? One of the other two? Maybe even the name, or the name of the color of the point! I would envision to have a sorting function that takes the command (or function) to be used for determining, what is the correct order for sorting. However, for this to work, not only type-safe containers would be necessary, but also being able to pass commands (or functions) as actual objects, which brings us directly to my next vision:

Commands and functions should be available as first level objects. Consider the following (again rather artificial) code:

```
function calculateValueOne(FDouble a, FDouble b) : FDouble
  return (a * b)
endfunction

function calculateValueTwo(FDouble a, FDouble b) : FDouble
  return (a + b)
endfunction

// pre: FBool argument "useFirst", objectlist "list" with a couple of points
objectlist results()
foreach (F3DPoint poi : list)
  if (useFirst)
    results.append(calculateValueOne(poi.getX(), poi.getY())
  else
    results.append(calculateValueTwo(poi.getX(), poi.getY())
  endif
endfor
```

```
foreach (F3DPoint poi : list)
  if (useFirst)
    results.append(calculateValueOne(poi.getZ(), poi.getY())
  else
    results.append(calculateValueTwo(poi.getZ(), poi.getY())
  endif
endfor
```

Again, I am very aware, that this code doesn't serve any meaningful purpose at all, it's just to illustrate my point and I am sure, you can think of real-life examples where similar things happen (not necessarily with functions, but this idea could be expanded to built-in commands as well). Now, wouldn't the following be much easier on the eyes (and yield better performance, too):

```
// Omitted the function declarations, they are the same as above
// pre: FBool argument "useFirst", objectlist "list" with a couple of points
objectlist results()
if (useFirst)
  SomeType func(calculateValueOne) // Store the function to call into a variable.
else                               // I used "SomeType" because the actual, spelled out type I do not
  SomeType func(calculateValueTwo) // know and would depend on the way this is implemented by us.
                                   // Being a C++ developer, I'd think something like
                                   // Command<FDouble(FDouble, FDouble)> may work. For you this
                                   // may seem way too cryptic, though, so I am open for suggestions
endif

foreach (F3DPoint poi : list)
  results.append(func(poi.getX(), poi.getY())
endfor

foreach (F3DPoint poi : list)
  results.append(func(poi.getZ(), poi.getY())
endfor
```

Now, to extend all of the above, I think it would be great to have **additional container** types available. I am talking about associative containers like maps, sets, or hashes. So, basically, I think we have some things to improve regarding containers, since they are so important for effective programming. Multi-dimensional lists/arrays would also be a very reasonable candidate to implement.

Another thing I would like to have is **universal copying and assignment**. Let's consider this code:

```
fp_myFeature c1() // create an instance of a FFeatureDefinition with the name "myFeature"
// initialize c1
fp_myFeature c2(c1)
```

The intention of this code is pretty clear, I think. Create a new Feature instance c2 that has the same attributes as c1. However, this is not what this code does. In fact, it will most likely not even compile (except if by any chance the first argument of `myFeature` is of type *FFeature::myFeature*). Creators that copy the values of the given object if that object has the same type as the object that is to be created are unfortunately not automatically present. In my opinion they should be, though (but it's not that easy to do, I must admit).

The assignment operator "`=`" is pretty much the same thing. Again, I think the intention of the following code should be self-explanatory:

```
double a(6)
double b = a
```

Create a new object *FDouble* "`b`" with the same value as the *FDouble* "`a`". But there is just no way to create a new object using the assignment operator. The only way to create new objects is by using a creator. Of course this would be easily overcome, by remembering to always use the creator command syntax if the intention is to create a new command, but this would only be possible in the presence of universal copy creators (there is one for *FDouble*, so `double b(a)` would work perfectly).

Assuming those universal copy creators exist for every type, you might say you couldn't care less for the assignment-creation syntax to work, but for me it just feels natural. However, the more important argument for universal assignment is the possibility to use it for on-the-fly changes of already existing objects to match the attributes of other objects. Currently the following will not compile:

```
BSplineCurve c1()
// initialize c1
BSplineCurve c2()
c2 = c1
```

Again, at least I think, the intention is completely clear: set all attributes of the (existing) *FBSplineCurve* `c2` to those of the *FBSplineCurve* `c1`. Unfortunately, this is currently not possible for objects of all types and the above code will not compile (actually, it's only possible for a very limited set of types).

Another interesting idea (at least for me) would be to offer the possibility to **derive Features from each other**. Now, I did give a very short introduction on object-orientation in the beginning, so I won't repeat it here and hope that you still remember it. In object-oriented programming classes can be related. For this idea the concept of base-classes and derived classes is relevant. Let's assume you implemented a nice Feature for your project. Now, let's further assume that it is applicable for many parts of the project but there are just some other parts where it does almost – but not exactly – fit. This "close but no cigar"-case could (sometimes) be handled by being able to say "I want to do exactly what that Feature does, but I want to do something beforehand or afterwards or I want a certain function to behave differently". This would be a case for "inheritance". Being able to say that the Feature B is the same as Feature A, it just needs to execute some additional code before or after A's code is – basically – the whole point of inheritance. Additionally, at least in my mind, it would be very beneficial for such a Feature-hierarchy to be able to not only add pre- and post-processing to the original Feature-code, but also to alter its behavior in between by offering alternate versions of functions in Feature B that were already defined in Feature A. Now, this would offer a whole new level to Feature-programming and would maximize the re-use capabilities of Features. Of course, for this to actually serve a purpose several things would have to be taken into account. There needs to be a way to initiate the execution of the original Feature in the derived one, functions need to be able to call their base-class implementations (similar to functions that hide global commands - see item 10 for more on this).

Another, thing that is even more illusionary, but is based on the current development regarding multithreading (I will not even try to explain this term, if you are completely unaware of the existence of that term, just skip this paragraph), is the idea of giving **multithreading capabilities to the Feature developer**. It would be great to give you (you are still reading this paragraph, so I have to assume, you know what I am talking about) the possibility to declare that a certain portion of your code should be executed in parallel, which would allow to utilize the capabilities of current CPUs. To be honest, though, I am rather sure that this idea will stay nothing but an idea, because the whole

concept of multithreading is so hard to wrap one's head around that I can hardly imagine to give this possibility to you in a - at least somewhat - safe way.

So, first of all, all of the above is nothing but my brain-fa***. So far there have been no (or close to none, i.e. maybe one or two) requests for any of those features. I hope you understand that we tend to not implement features that a) only benefit few people and b) cannot be sold to potential new customers. So, this means that the implementation of all of the above would be subject to some (potentially many) "google-fridays" for me. So, I'd be very interested to know: what are your thoughts? What language features are you missing (if any)? How do you like my ideas? Would you like to see something didn't even think of? Please comment in the blog post you got this from. If you didn't receive this file from one of our blogs, just write an email to support@friendship-systems.com or write in our forum or the helpdesk, it is much appreciated.

Bye, Bye!

For those of you that actually did stick around until here, thanks for reading! I hope you learned something new and had as much fun reading this as I had writing it (see below). I'd also like to offer you a (virtual) cookie for making it to the end (or two... heck take as many as you'd like).

One more thing and it's a thing I would like to have a cookie for: believe it or not, but I typed this whole thing on my smartphone during my train-rides to and from work. I must admit that I added formatting later (and got rid of some typos that inevitable are bound to happen when using the limited phone keyboard and trying to get a large amount of text down), but the text itself is written 100% using the touch-keyboard of my trusty phone (I will not tell which make or model, because I don't feel like advertising anything - if those people call and make the right offer, I will change this paragraph). So this whole thing was a nice “time-waster” on my (often troublesome) rides with the Berlin S-Bahn. If you did enjoy this (or didn’t enjoy it but learned something or you just want to reward a lunatic that wrote this much text on a phone) I would be happy if you could upvote the blog-post you’ve downloaded this from. I (unfortunately) do not get a commission for those upvotes, but it makes me happy.

Anyways, now it's really time to say "good-bye". Thanks again for reading!