Operating

The newer and more robust way of handle your chains is through the process called operating.

C++ Workflow

Using a Lambda

You can easily operate on your chain via a C++ lambda and this is how you do it:

```
Chain->Operate([](FMyTrait Trait)
{
    ...
});
```

Note that you're not allowed to acquire a reference to the trait while processing a non-solid chain, only its copy. So in order to operate on a solid chain, you could do something like this:

```
SolidChain->Operate([](FMyTrait& Trait)
{
    ...
});
```

Now you can change the properties (fields) of the trait directly, without copying involved.

Concurrency

Solid Chains also support a special type of operating - a multi-threaded one. The function to call is explicitly named with a `Concurrently` prefix and accepts two more arguments: the maximum number of tasks to utilize and the minimum number of slots per each such task. For example:

```
SolidChain->OperateConcurrently([](FMyTrait& Trait)
{
    ...
}, 4, 32);
```

The second parameter helps to also limit the number of tasks. If there are too little slots available, excessive tasks not needed for that quantity won't be queued at all.

Dependency Injection

One great thing about operating is that the function arguments are actually resolved and delivered automatically to your logic. For example, if you also modify the currently iterated subject, just specify the Subject handle in the very declaration of the routine:

```
Chain->Operate([](FSubjectHandle Subject, FMyTrait Trait)
{
    ...
});
```

This, of course, has to match the solidity of the chain. So for a solid chain this would be:

```
SolidChain->Operate([](FSolidSubjectHandle Subject, FMyTrait& Trait)
{
    ...
});
```

You can actually ask for different contextual information within the loop. For example:

```
Chain->Operate([](const FChain* Chain, const FChainCursor& Cursor,
ISubjective* Subjective, FMyTrait Trait, UMyDetail* Detail)
{
    ...
});
```

Current Iteration Index

Just in case you need to know the number of the current iterated Slot (i.e. Subject's place within the chain), use the dedicated GetChainSlotIndex() method of the corresponding Cursor type, which can also be delivered using the aforementioned means:

```
SolidChain->Operate([](FPlacementTrait& Placement, const FSolidChainCursor&
Cursor)
{
    Placement.Number = Cursor.GetChainSlotIndex();
});
```

Stopping

While conceptually not very clean it is sometime useful to stop the actual processing (iterating) of the chain prematurely, manually. This can be easily accomplished with a dedicated method.

For example:

```
int32 Counter = 0;
Chain->Operate([&Counter](const FChain* Chain, FMyTrait Trait)
{
    if (Counter > 100)
    {
        Chain->StopIterating();
        // Return explicitly, so the counter doesn't get incremented on the
current iteration:
        return;
    }
    Counter += Trait.Value;
});
```

Direct Mechanism Operating

The Operating simplification goes as far as actually Operating on a Mechanism directly, like so:

```
Mechanism->Operate([](FGlowing Glowing, FHelmet Helmet)
{
    ...
});
```

This way the Filtering and the Enchaining is done automatically under the hood, deriving the necessary Components from the lambda arguments.

You can of course supply the Filter specification explicitly, overriding it.

So for example, if you want to specify an additional Trait and a Flagmark condition, do it like so:

```
Mechanism->Operate(FFilter::Make<FGlowing, FHelmet, FHero>(FM_Z),
[](FGlowing Glowing, FHelmet Helmet)
{
    ...
});
```

The direct Operating mode is smart enough to deduce the type of the Chain used within the Operating process, so if you specify a reference to a Trait and/or a Solid Subject handle in your arguments list, the technique will essentially produce a Solid iterating:

```
Mechanism->Operate([](FSolidSubjectHandle Subject, FGlowing& Glowing,
FHelmet& Helmet)
{
    ...
```

```
});
```

You can still specify the Chain type explicitly as a first template argument to a method:

```
Mechanism->Operate<FSolidChain>([](FGlowing Glowing, FHelmet Helmet)
{
    // You logic within Solid semantics:
    ...
});
```

Concurrency variants are also provided by the direct interface:

```
Mechanism->OperateConcurrently([](FSolidSubjectHandle Subject, FGlowing&
Glowing, FHelmet& Helmet)
{
    ...
}, /*Maximum number of threads=*/4, /*Minimum number of Subjects=*/16);
```