Does an Ambient Property define a Requirement?

As of October 1, 2012 (morning), the order was:

1. All MutableItem creation:
   1. For each StObj path to a concrete class:
      1. Creation of the Structured Object instance (the leaf type of the path), by Activator.CreateInstance on the default constructor.
      2. A MutableItem is created for each Type in the path (from root to leaf).
      3. The MutableItem is configured from leaf to root.
         1. Information at type level is applied (attributes). Container, Generalization, Requires, RequiredBy are initialized.
         2. Construct method is analyzed. Requirements are updated based on parameters.
         3. Attributes that support IStObjStructuralConfigurator are solicited.
         4. External IStObjStructuralConfigurator can configure the MutableItem.
2. Ordering
   1. For each MutableItem, Typed dependencies (container, requires, etc.) are resolved to their associated MutableItem.
   2. Calls DependencySorter.OrderItems to sort the MutableItems.
3. Initializing
   1. For each MutableItem (sorted) resolve ambient properties.

This currently uses the fact that the DependencySorter handles Container inheritance through the Generalization reference…

* 1. For each MutableItem (sorted) call Construct methods.

Depending on StObjCollector.CallConstructBeforeResolvingProperties, Ambient Properties are resolved before Construct method is called – the default – or after (invert 3.a and 3.b above).

In both cases, resolving ambient properties requires Container/Generalization chains to be exploited: resolving after ordering eases the job (no cycle).

If we want Ambient Properties to participate in dependency order they must be resolved earlier, before the sort but after the MutableItems binding: between 2.a and 2.b.  
The CallConstructBeforeResolvingProperties is no more a viable option… We simply remove it.

We can no more rely on Container inheritance provided by the DependencySorter. Instead, we must resolve the Container inheritance by following the Generalization chain during Ambient Properties resolution. This does not impact too much the current implementation since Ambient Properties resolution already handles reentrancy…

As of October 1, 2010 (evening), the order is:

1. All MutableItem creation:
   1. For each StObj path to a concrete class:
      1. Creation of the Structured Object instance (the leaf type of the path), by Activator.CreateInstance on the default constructor.
      2. A MutableItem is created for each Type in the path (from root to leaf).
      3. The MutableItem is configured from leaf to root.
         1. Information at type level is applied (attributes). Container, Generalization, Requires, RequiredBy are initialized.
         2. Construct method is analyzed. Requirements are updated based on parameters.
         3. Attributes that support IStObjStructuralConfigurator are solicited.
         4. External IStObjStructuralConfigurator can configure the MutableItem.
2. Handling dependencies
   1. For each MutableItems (starting from Specialization), we *Prepare* it:
      1. Typed dependencies (container, requires, etc.) are resolved to their associated MutableItem.
      2. Recursive calls:
         1. Generalization is *Prepared*.
         2. Container is *Prepared* or inherited (if actual container is null and Generalization exists)
      3. Ambient Properties are located and set.
   2. Calls DependencySorter.OrderItems to sort the MutableItems.
3. Initializing
   1. For each MutableItem (sorted) call Construct methods

Ambient Properties can now participate in dependencies.

As of start of December, this has changed with the introduction of StObjProperties.

As of December 4th, 2010, the whole process is:

* AssemblyRegisterer

Discovers types in assemblies.

* StObjCollector

Starts from registered types and produces an ordered list of IStObj (slice of object ordered by their dependencies) inside one or more Context.

* StObjSetupBuilder

Starts from ordered lists of IStObj in one or more Contexts and produces a collection of ISetupItem.

* SetupCenter

Starts from multiple sets of IDependentItem, sorts them, associates a SetupDriver to each of them and executes the Init/Install/Settle steps on the drivers.

|  |  |  |
| --- | --- | --- |
| Character | Unicode value | CycleExplainedElement constant & Description |
| ↳ | \u21B3 | **Start** : Start of list. |
| ∈ | \u2208 | **ElementOf** : Item A belongs to Group G. |
| ∋ | \u220B | **Contains** : Group G contains an Item A. |
| ⇀ | \u21C0 | **Requires** : Item A requires Item B. |
| ↽ | \u21BD | **RequiredBy** : Item A is required by Item B. |
| ⇌ | \u21CC | **RequiresRequiredBy** : Item A requires Item B because B is required by A. |
| ⊏ | \u228F | **ElementOfContainer** : Item A belongs to Container C. |
| ⊐ | \u2290 | **ContainerContains** : Container C contains Item A. |
| ↟ | \u219F | **GeneralizedBy**: Item A is generalized by Item B. |

Context-Location naming.

Eventually on November 17, 2012, projection from objects participating to setup into dependency FullName namespace seems on the right way… Its name is “ContextLoc”. (and it took me too much time to exhibit it!).

A projected dependent item FullName is made up of 3 parts:

- Context: is the object scope for Dependency Injection and pseudo-singleton (Structured Object layer). IAmbientContract objects and interfaces instances are unique inside a Context. There is no subordination between Contexts: a simple name (a string) is enough to define and identify a Context. A default Context exists that is identified by the empty string. Null Context corresponds to an unknown, undefined context.

- Location: is the logical location of the object. It describes a “container” for the object; this is typically the logical name of a database (“db”, “dbHisto”, etc.) or a computer name, a sub-network, etc.  
This location has nothing to do with Context: these are totally orthogonal concepts. Location should support a kind of hierarchical naming structure and a way to express reference in a relative manner (location should be combinable like paths are). The empty string designates the “root” of the namespace. Null Location corresponds to an unknown, undefined location (just like Contexts).

- Name: is the unique name of the objet inside its Location. There should be the less possible restrictions on allowed characters in an object Name. The empty name (empty string) may exist; null name has no existence.

Default implementation is in DefaultContextLocNaming and is currently exposed as static methods. Whenever multiple/different syntaxes will be required because of limitations of this default implementation, a IContextLocNaming interface and multiple implementations should be created and the IContextLocNaming injected in the architecture wherever it is required (this would be a breaking change).

ContextLocNaming default implementation

General syntax is the following one:

**[**Context**]**loc-subloc-subloc2**^**Object.Name.Comes.Here

Rules are:

* Null Context 🡺 “” (empty string)
* Default Context 🡺 “[]”
* Named Context 🡺 “[NamedContext]”
* Null Location 🡺 “” (empty string)
* Root location 🡺 “^“
* Location 🡺 “loc^”
* Location Path 🡺 “loc1-loc2-loc3^”
* Valid Names 🡺 Must not start with “[“ nor contains “^“.

Parsing examples:

|  |  |  |  |
| --- | --- | --- | --- |
| Input | Context | Loc | Name |
| CK.fTest | Null (Unknown Context) | Null (Unknown Location) | CK.fTest |
| []CK.fTest | Empty (Default Context) | Null | CK.fTest |
| []^CK.fTest | Empty | Empty (Root Location) | CK.fTest |
| ^CK.fTest | Null | Empty | CK.fTest |
| [X]^CK.fTest | X | Empty | CK.fTest |
| [X]db^CK.fTest | X | db | CK.fTest |
| [X]-db^CK.fTest | X | -db | CK.fTest |
| [X]--db^CK.fTest | X | --db | CK.fTest |
| [X]---srv-db^CK.fTest | X | ---srv-db | CK.fTest |
| srv-db^CK.fTest | Null | srv-db | CK.fTest |
| [X]srvPrd-db^CK.fTest | X | srvPrd-db | CK.fTest |
| []srvPrd-db^CK.fTest | Empty | srvPrd-db | CK.fTest |
| Empty string | Null | Null | Empty string |
| - | Null | Empty | Empty string |
| []^ | Empty | Empty | Empty string |
| [X]loc^ | X | loc | Empty string |

Unknwon Context and Location can be initialized by the current context. The current context is defined by curContext and curLoc (two strings that may be null or empty). Below are examples of calling method Resolve(input, curContext, curLoc ):

|  |  |  |  |
| --- | --- | --- | --- |
| input | curContext | curLoc | Result |
| CK.fTest | Null | Null | CK.fTest |
| CK.fTest | Empty | Null | []CK.fTest |
| CK.fTest | Empty | Empty | []^CK.fTest |
| CK.fTest | Null | Empty | ^CK.fTest |
| CK.fTest | X | db | [X]db^CK.fTest |
| ^CK.fTest | X | Nimp | [X]^CK.fTest |
| srv-db^CK.fTest | X | Nimp | [X]srv-db^CK.fTest |
| []srv-db^CK.fTest | nimp | Nimp | []srv-db^CK.fTest |
| []^CK.fTest | nimp | Nimp | []^CK.fTest |

To support “Location as a path”, Resolve method can combine Location, following the syntax below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Input | curContext | curLoc | Result | Comment |
| -^CK.fTest | Null | srv-db | srv^CK.fTest | Leading “-“ goes up the location path... |
| --srv2-db3^CK.fTest | Empty | sys-srv-db | []sys-srv2-db3^CK.fTest | …and can be combined with sub path. |
| ---srv2-db3^CK.fTest | Empty | Null | []---srv2-db3^CK.fTest | Of course, when curLoc is Unknown, location is left unchanged. |
| -^X | Nimp | Empty | Error. | Going up the path above the root location is an exception. |
| --^X | Nimp | oneLoc | Error. |

Derived names introduce a structure in names. Their goal is to support both transformations and compositions

Composition may be as simple as using the dot operator (or any other separator) to identify the final dependent item from any parent container. Question: do we need to capture the type of the final item?

Transformations are defined by an item and act on another one. There seem to be no need to support multiple transformations of the same item for the same definer: required transformations from a site SHOULD be grouped (the “impact” of item A on B is defined once).

This (claimed) unicity allows us to easily name a transformation with parentheses: transformer(transformed). Should a transformation apply on any possible object (i.e. the transformed item FullName should be used) or should the transformed object necessarily belong to the same Context or Location (i.e. LocName or Name only should be used)?

There is no issue to allow any possible item to be transformed and I can see no advantage to restrict it.

A typical transformation name is: []db^CKLevel0.Package([]db^CK.sTest)

A dependent item can be an Object or a Transformation (and a Transformation is an Object).

Questions:

* Can a transformation applied to a type T return a type T’?

Definitely not. A view must remain a view and a stored procedure a stored procedure.

* Can we transform a Transformation?

Ideally yes. This leads to a fundamental issue: the application ordering (and times) of transformation processes. By using a Composite, we can have a generic, global answer to this issue

SetupObjectItem (origin)

Transformations[]

Final 🡺 Transformations.Last.Final

A Transformation contains its SetupObjectItem.

StObjProperties are not like AmbientProperties and AmbientContracts.

* StObjProperties are bound to a StObj (a slice of the Structured Object).
  + They are normally not applicable to the Structured Object itself. If they are, they MUST be declared with a mask (the C# “new” keyword) on each and every class that specialize the one that defines the property (and of course it must not be virtual).
  + When not locally declared (explicitly on the StObj), the value is searched on Containers first and then on Generalization.
* AmbientProperties and AmbientContracts are properties of a Structured Object itself.
  + What they have is common:
    - Only one value can exist for a given property among the specialization chain.
    - They support covariance: any specialization can redefine the property with a more specialized type than the one of its base class.
    - The Ambient Contract (or Property) can be marked as Optional (on the attribute), but is Required by default.
      * Optionality is “inherited” but under control. If a base class defines an Optional Ambient Contract (or Property), a specialized implementation can explicitly require it. As soon as an Ambient Contract (or Property) is defined as Required, none of its specialization can declare it as Optional.
  + The difference between the two is:
    - Ambient Contracts are bound to IAmbientContract objects that will be injected at the end of the whole resolution.
      * When not marked as Optional, if the IAmbientContract can not be resolved, an error is raised.
    - Ambient Properties can be (typically) IAmbientContract objects but may be of any type.
      * When not set (explicitly on a StObj), the value is searched, by default, on Generalization and then on Containers: AmbientProperties are first inherited (from Generalization) and then initialized from Containers.  
        This can be changed on the attribute :

[AmbientProperty( ResolutionSource = PropertyResolutionSource.FromContainerAndThenGeneralization )]  
public string OneStringValue { get; set; }

* + - * They can participate in ordering since the referenced type (when it is also a Structured Object) can decide that all StObjs that reference it be considered as Children, or Requirements, etc. (thanks to TrackAmbientPropertiesMode).
      * They are resolved before calling Construct (to enable them to participate to ordering). Depending on the property, we will be able to set it before Construct or only after the whole graph is created.
        + When the AmbientProperty is a mere value (not a StObj), the value will be set before Construct.
        + When the AmbientProperty is a StObj, depending on the resolved StObj's TrackAmbientPropertyMode, we set the value before Construct if and only if the relationship guaranties that the target StObj (not necessarily its specialization) will be constructed before this object.

**Idea:** to better handle template method pattern, Construct methods on non-ambient contract base classes should be considered as “a part” of the Construct of the top ambient class:

class SqlDatabase { void Construct( string connectionString ) {…} }

class SqlHistoDatabase : SqlDatabase { void Construct( IOriginAccessor origin ) {…} }

One should consider that the SqlHistoDatabase Construct to be Construct( string connectionString, IOriginAccessor origin ), and call the base classes Construct with their respective parameters.

Overall assemblies’ architecture

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Dependency Injection & Configuration of object graphs.**  Type discovering, dependencies analysis & graph configuration. Final objects are obtained from a dynamically emitted assembly. | **Three Setup Steps support.**  Setupable objects support: version management, scripts and three steps setup (Init/Install/Settle). | **Sql Server objects.**  Sql Server setupable objects like Tables, Procedures, etc. |
| Engine level: analyses assemblies, discover types, build and configure the final objects. Emits the final compiled dll. | CK.StObj.Engine | CK.Setupable.Engine | CK.SqlServer.Setup.Engine |
| Optional: defines objects and attributes that describes how to build and configure final objects. | CK.StObj.Runtime | CK.Setupable.Runtime | CK.SqlServer.Setup.Runtime |
| Required: contains core objects and attributes definitions that final objects require. | CK.StObj.Model | CK.Setupable.Model | CK.SqlServer.Setup.Model |

SqlTokenType bit flags

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1** | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

Error or EndOfInput (negative values)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1** | **0** | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

EndOfInput

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1** | **1** | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

Error

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1** | **1** | **1** | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

ErrorTokenizerMask

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

SqlTokenType.None (zero)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 30 | 29 | **28** | **27** | **26** | **25** | **24** | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

Operator precedence bits n°28 to 24  
n°28 currently unused. 15 operator levels used.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 30 | 29 | 28 | 27 | 26 | 25 | 24 | **23** | **22** | **21** | **20** | **19** | **18** | **17** | **16** | **15** | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

Token discriminators bits n°23 to 15 (IsAssignOperator to IsComment).  
One and only one bit is set among them except for operators that are also identifiers (logical and select operators):

|  |  |  |
| --- | --- | --- |
| 23 | IsAssignOperator | Covers =, |=, &=, ^=, +=, -=, /=, \*=, and %=. |
| 22 | IsBasicOperator | Covers binary operators |, ^, &, +, -, /, \*, % and the unary ~ (bitwise not). |
| 21 | IsBracket | Covers [], () and {}. |
| 20 | IsCompareOperator | Covers =, >, <, >=, <=, <>, !=, !> and !<. |
| 19 | IsIdentifier | Covers identifiers. |
| 18 | IsNumber | Covers binary, money, float and integer (hexadecimal). |
| 17 | IsPunctuation | Covers dot ".", comma "," and semicolon ";". |
| 16 | IsString | Covers strings ('string' or N'string'). |
| 15 | IsComment | Covers /\* ... \*/ block as well as -- line comment. |

Identifier specific:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | **19** | 18 | 17 | 16 | 15 | **14** | **13** | **12** | **11** | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

IdentifierTypeMask (bits n°14 to 11) – 16 possible types - 9 identifier types are defined:

|  |  |  |
| --- | --- | --- |
| 0 | IdentifierStandardStatement | Not reserved keywords that can start a statement like “get”, “move”, “receive”. |
| 1 | IdentifierReservedStatement | Reserved keywords that starts a statement: “select”, “create”, “declare “set”, etc. |
| 2 | IdentifierStandard | Any identifier like “max”, a table name, but not a keyword like keyword like “when”, “select” or “else”. |
| 3 | IdentifierReserved | Identifiers that are reserved keywords (like “identity\_insert”, “clustered”, “rule”, “as”, etc.) but cannot start a statement. |
| 4 | IdentifierQuoted | Denotes a "quoted identifier". |
| 5 | IdentifierQuotedBracket | Denotes a [Quoted identifier]. |
| 6 | IdentifierSpecial | Special identifiers like star (in “select t.\* from t), $identity, $Partition, etc. |
| 7 | IdentifierDbType | SqlDbType like int, smallint, datetime, xml, etc. |
| 8 | IdentifierVariable | @myVariableName or @@SystemFunctions like @@RowCount or @@Error. |

Handling “with schemabinding” option

Facts about schema bound objects (views, functions and even Natively Compiled Stored Procedures):

* You can not change the collation of a database with schemabound objects.
* You can not use SELECT \* in a schemabound view.
* You can not run sp\_refreshview on a schemabound view. You do get a rather unhelpful error though.
* You can make any change to the table that do not affect the structure of the bound columns.
* You can find out if an object is schemabound by looking at the column is\_schema\_bound in sys.sql\_modules or the system function OBJECTPROPERTY(object\_id, ‘is\_schema\_bound’).
* If you reference a view or function in a schemabound view or function then that view or function must also be schemabound.
* Objects that are bound (tables/views) can not be dropped while a schemabound object references them

SB objects form a “stronger” DAG in terms of dependencies. Once a SB exists, any transformation to its dependencies are impossible. We can consider that the TYPE objects introduce a similar “strong dependency”: altering a TYPE definition requires a drop/create and this can be made iif the TYPE is not used at all by any other objects: column of a table, parameter, local variables…

We DO NOT use TYPE for columns (see <http://stackoverflow.com/questions/1383494/alter-user-defined-type-in-sql-server> and <http://www.sql-server-performance.com/2008/how-to-alter-a-uddt/>). However we may use it for parameters and local variables and, actually, NOT for scalar types but only for table types.

One way to work around the Table Type complication may be to consider what we can call a “inline table type” (ITT). ITT would allow us to define a table type more like a macro, or an alias, than an actual type: wherever we need it, we write the full definition of the type and this definition is transformed in a type name associated to its definition. Obvious drawbacks: we do not write valid T-Sql anymore, actual T-Sql may be cryptic and this is not that easy to come with a name mangling that satisfies readability and offer us a correct “bijection” between names and actual definitions…

If we don’t follow the previous path, we must deal with the drop/create and accept a fact: we’ll need to drop all objects that use the type before altering it.

This is not the current way CK-Database works: objects are altered or drop/created in the *install* step. This introduces a pre step of drop for some objects in the database that may occur in the *init* step of the process.

What are the objects that need to be dropped in the *init* and be created in the *install* step? It is obvious: all the objects that “strongly depend” on any object that will be altered.

This raises an important issue: how do we know that an object is (will be) altered? The answer is that we currently don’t know: object level versioning has exploded with the advent of the “transformers”, there is at this time no replacement for this lost version… And this is a rather sensible issue since this is required to support “strong dependencies” but, more importantly, should allow a real optimization of the setup process and enables interesting “diff” scenarios in CK-Database…

Object versioning should now be elected as one of the top next features to work on.

Given this Object Versioning capability, is it possible to handle this correctly at the object level?

Unfortunately, not always. When a view references a table, altering the table may fail or succeed depending on the actual columns of the table that are involved. We may consider that any migration script attached to a table is a change, but what about the migration scripts that may exist in packages or any other tables?

The only trivial case is when absolutely no migration scripts (Model scripts) must run. Unfortunately, there are the “Always scripts”. Those scripts should be idempotent, and this does mean that they sometimes do something: nothing prevents such an “Always Script” to actually contain a one-shot migration script (“if not exists column then create it”). This lead us to consider that this trivial case should be ignored in practice: we should try to detect impacts on the model objects (i.e. the tables).

There are two ways to detect any impact in the schema:

* By analyzing the script itself. This is not the way to go, this is risky (false positive/negative) and complex (requires a deep understanding of the script language).
* By running the script and comparing the object before and after. This can be easily done with a hash of the two objects’ definition in the *install content* step of the “Model” root object.

The problem with the second proposal is that we need to know that something WILL change, not HAS changed: the objects that “strongly depend” (schema bound view) on the altered object (the table) must be dropped before.

There seems to be no obvious way to avoid the ultimate solution that is to drop all schema bound views at *init* step (before migration takes place in *install* step).

This lead to a system where scripts and objects are deeply different:

* Actual schema objects (tables) in the Model are altered by scripts. These scripts being versioned.
* All other objects are destroyed and recreated.

This should enforce a principle that is currently not obvious: tables must not depend on objects. Tables have one and only one responsibility: to hold the data. Computed columns may exist but only with trivial computations, any non-trivial functional dependencies must be handled by views and functions objects.

Focusing on objects, we can change the (no more applicable) version management with a content hash based mechanism:

* Objects are registered during their installation in a core table associated to their hash.
* During setup, objects creation scripts are created in memory and their hashes are computed.

Are the objects for which hash differs the objects that must be recreated? This is the case for non-schema bound objects, but for schema-bound objects we must consider its dependencies, and not its current ones but the dependencies of the previous setup.

One way to ease this process is to consider the hash of the dependencies when computing the hash of an object (a kind of Merkel Tree).

First step is to handle “strongly dependent objects”. That means that a dependent item must be able to require that its dependencies are fully transformed: it actually depends on the object’s transform target if one exists.

First, let’s consider the transitivity of this “strong requires”. When a “strong dependent object” depends on a package, does it mean that all objects in the package have to be transformed first?

1. No, because it would put too much pressure on the graph ordering: one such object can unnecessary freeze a whole set of objects. Such strong dependencies should only remain at the objects level, directly expressed by the object’s explicit dependencies to other objects. It is an ad hoc mechanism for schema bound objects rather than a general one.
2. Yes. This is an intrinsic property of the object to setup. All its dependencies (including all transitive ones) must be fully built and setup.  
   Because of the RequiredBy relationships, this has to be handled by the dependency sorter. Current implementation manages to maintain/update the transform target during graph build: objects that are transformed know their transformed target and expose it (the transform target depends on its source). This target must be accounted by the topological sort as a new kind of relationships between objects. A new RequiresMustBeTransformed boolean property on any item must drive the topological sort: when true, it is the transformed object that must be considered as the actual requirement.

The first approach is obviously easier to implement since this does not impact the sorter and its weakness is protected by the target system itself (a schema bound object requires its dependencies to also be schema bound). In this scenario, RequiredBy declarations are not like actual direct dependencies (they are more like a relaxed “before that” requirement) and this seems fine (at least for the moment) since a RequiredBy is a statement from the dependent object, it is not a “true” dependency of the object itself.

This second approach currently seems the best way.