Aspect-Oriented Modeling

The technique of *Aspects* provides a very powerful means to organize your models in a way that keeps your core domain models concise and comprehensible by factoring out secondary concerns into separate files, defining and reusing common aspects, as well as adapting reused definitions to specific needs.

↳ ***See also:*** *Respective section in Five reasons to use CAP , and Separating concerns and focusing on important stuff blog posts by DJ Adams.*

Table of Contents

- Avoid: Table Per Leaf Class Strategy
- Avoid: Table Per Class Strategy
- Prefer: Single Table Strategy

Similar to Aspect-Oriented Programming

Aspect-oriented Modeling as promoted by CDS is very similar in goals and approaches to Aspect-oriented Programming as defined in this Wikipedia article :

> *Aspect-oriented programming (AOP) is a programming paradigm that aims to increase modularity by allowing the **separation** of **cross-cutting concerns**. It does so by adding behavior to existing code (an **advice**) without modifying the code, [...].*

> **Extend anything from anywhere**
>
> In essence CDS Aspects allow you to arbitrarily spread a definition across different places in the same files, or separate ones, in different projects, with different ownerships and different lifecycles.

Separation of Concerns

Use aspects to factor out secondary concerns into separate files as follows...

Avoid: All-in-one Models

Instead of polluting your core domain models with a multitude of annotations, put such annotations into separate files. For example, instead of having a single-source model like that:

srv/cat-service.cds

cds

```
service CatalogService {
  @UI.SelectionFields: [
    ID, price, currency_code
  ]
  @UI.LineItem: [
    { Value: ID, Label: '{i18n>Title}' },
    { Value: author, Label : '{i18n>Author}' },
    { Value: genre.name},
    { Value: price},
    { Value: currency.symbol},
  ]
  @UI.HeaderInfo: {
    TypeName       : '{i18n>Book}',
    TypeNamePlural : '{i18n>Books}',
    Description    : { Value: author }
  }
  @UI.HeaderFacets: [{
    $Type  : 'UI.ReferenceFacet',
    Label  : '{i18n>Description}',
    Target : '@UI.FieldGroup#Descr'
  }]
  @UI.Facets: [{
    $Type  : 'UI.ReferenceFacet',
    Label  : '{i18n>Details}',
    Target : '@UI.FieldGroup#Price'
  }]
  @UI.FieldGroup #Descr : { Data: [{Value : descr}, ]}
  @UI:FieldGroup #Price : { Data: [
    { Value: price},
    { Value: currency.symbol, Label: '{i18n>Currency}' },
  ]}
  entity Books { ... }
  ...
}
```

Prefer: Keep Your Core Clean

Rather, keep your core model concise and comprehensible:

srv/cat-service.cds

```cds
service CatalogService {
  entity Books { ... }
  ...
}
```

Prefer: Factor Out Separate Concerns

And factor out the UI concerns into a separate file like that:

app/fiori-layout.cds

```cds
using { CatalogService } from '../srv/cat-service';

// Annotations for List Pages
annotate CatalogService.Books with @UI:{
  SelectionFields: [
    ID, price, currency_code
  ],
  LineItem: [
    { Value: ID, Label: '{i18n>Title}' },
    { Value: author, Label : '{i18n>Author}' },
    { Value: genre.name},
    { Value: price},
    { Value: currency.symbol},
  ]
}

// Annotations for Object Pages
annotate CatalogService.Books with @UI:{
  HeaderInfo: {
    TypeName       : '{i18n>Book}',
    TypeNamePlural : '{i18n>Books}',
    Description    : { Value: author }
  },
  HeaderFacets: [{
    $Type  : 'UI.ReferenceFacet',
    Label  : '{i18n>Description}',
    Target : '@UI.FieldGroup#Descr'
  }],
  Facets: [{
    $Type  : 'UI.ReferenceFacet',
```

```
    Label  : '{i18n>Details}',
    Target : '@UI.FieldGroup#Price'
  }],
  FieldGroup #Descr : { Data: [{Value : descr}, ]},
  FieldGroup #Price : { Data: [
    { Value: price},
    { Value: currency.symbol, Label: '{i18n>Currency}' },
  ]}
}
```

Common Reuse Aspects

Quite frequently, you want some common aspects to be factored out and shared by and applied to multiple entities. For example, lets assume we'd want to factor out the common aspects of a standardized primary key, managed data, change tracking, extensibility, and temporal data...

Avoid: *Max Base Class* Approach

The classic way to do so, for example in class-based inheritance systems like Java, is to have a central team defining single base classes like `Object` for that, and either add all the aspects in question to that single base class, or have a base class hierarchy, like that:

cds

```
abstract entity BusinessObject {
  key ID     : UUID;
  createdAt  : DateTime;
  createdBy  : User;
  modifiedAt : DateTime;
  modifiedBy : User;
  changes    : Composition of many Changes;
  extensions : PredefinedExtensionFields;
}
```

▶ *With `Changes` and `PredefinedExtensionFields` defined like that...*

```cds
abstract entity TemporalBO : BusinessObject {
  validFrom  : Date @cds.valid.from;
  validTo    : Date @cds.valid.to;
}
```

Consumers would then use these base classes like that:

```cds
using { BusinessObject, TemporalBO } from 'your-base-classes';
entity Foo : BusinessObject {...}
entity Bar : TemporalBO {...}
```

> **Issues with that approach...**
>
> One issue is that due to single inheritance limitations, these base classes frequently have to combine several actually independent aspects into one definition, and the consumers have to take them all. Related to that is that these base classes have to depend on each other, which ultimately means they can only be provided and owned by central teams.

▶ *`abstract entity` is deprecated...*

Prefer: Separate Reuse Aspects

While, as shown above, the central single-inheritance-style base class approach is also possible with CDS, we can do better using CDS Aspects, leveraging the equivalent of multiple inheritance, and hence distributed ownership instead of central one:

```cds
aspect cuid { key ID : UUID; }
```

```cds
aspect managed {
  createdAt  : DateTime;
  createdBy  : User;
  modifiedAt : DateTime;
  modifiedBy : User;
}
```

```cds
aspect tracked {
  changes : Composition of many Changes;
```

```
}
```

cds

```
aspect extensible {
  s1 : String;
  s2 : String;
  s3 : String;
  i1 : Integer;
  i2 : Integer;
  dt1 : DateTime;
  ...
}
```

cds

```
aspect temporal {
  validFrom : Date @cds.valid.from;
  validTo   : Date @cds.valid.to;
}
```

↳ *Some of such common reuse aspects are already covered by `@sap/cds/common`.*

Consumers would then flexibly use these reuse aspects like so:

cds

```
using { cuid, managed, tracked, extensible, temporal } from 'your-reuse-aspec
entity Foo : cuid, managed, tracked, extensible {...}
entity Bar : cuid, managed, temporal {...}
```

Advantages of that approach

Not only does that approach allow clearer separation of concerns, and thus freedom of choice on which combinations of aspects to pick for consumers, it also allows distributed ownership of such reuse aspects, as they don't depend on each other.

Looks Like Inheritance...

The `:` -based syntax for includes looks very much like (multiple) inheritance and in fact has very much the same effects. Yet, it is not based on inheritance but on mixins, which are more powerful and also avoid common problems like the infamous diamond shapes in classical inheritance-based approaches.

Adaptation of Reused Definitions

Assumed there's a reuse package offering some common types and entities which would nicely fit your needs. For example:

some-reuse-package/index.cds

```cds
entity Currencies : CodeList { key code : String(3); }
entity Countries : CodeList { key code : String(5); }
entity Languages : CodeList { key locale : String(5); }
type CodeList : {
  name : localized String;
}
```

Adding / Adapting Fields — *Best Practice*

Now also assumed, you'd want all code lists to have an additional field for long descriptions, and you also want currency symbols, and the `locale` field for languages needs to support values with up to 15 characters. With aspects, you could simply adapt the reuse types and entities accordingly as follows:

db/common.cds

```cds
using { CodeList, Currencies, Languages } from 'some-reuse-package';
extend CodeList with { descr: localized String }
extend Currencies with { symbol: String(2) }
extend Languages:locale with (length:15);
```

Adding Relationships — *Best Practice*

You can even add Associations and Compositions to definitions you obtained from somewhere else. For example, the following would extend the common reuse type `managed` obtained from `@sap/cds/common` to not only capture latest modifications, but a history of commented changes, with all entities inheriting from that aspect, own or reused ones, receiving this enhancement automatically:

```cds
using { User, managed } from '@sap/cds/common';
extend managed with {
  ChangeNotes : Composition of many {
    key timestamp : DateTime;
    author : User;
    note : String(1000);
  }
}
```

↳ *Learn more about `managed` and `@sap/cds/common`*

Adding Reuse Aspects — *Best Practice*

And as the `:` notation to *inherit* an aspect is essentially just syntactical sugar for extending a given definition with a *named* aspect, you can also adapt a reused definition to *inherit* from a common reuse aspect from 'the outside' like so:

```cds
using { SomeEntity } from 'some-reuse-package';
using { managed } from '@sap/cds/common';
extend SomeEntity with managed;
```

Customization, Verticalization

The same approach and techniques are used by SaaS customers when customizing a SaaS application to tailor it to their needs.

Adding Custom Fields — *Best Practice*

For example, SaaS customers would quite frequently add extension fields like that:

```cds
using { ShipmentOrders } from 'some-saas-application';
extend ShipmentOrders with {
  carrier : Association to Carriers; // new association
```

```
  delayedBy : Time; // new field
}
```

↳ *Learn more about Extensibility*

Overriding Annotations — *Best Practice*

Sometimes they'd need to override existing annotations, such as for UI labels:

```cds
using { Customers } from 'some-saas-application';
annotate Customers with @title:'Patients'; // e.g. for health care
```

Verticalization — *Best Practice*

Verticalization means to adapt a given application for different regions or industries, which can be accomplished by providing respective predefined extension packages and switch them on per customer using feature toggles.

Inheritance Hierarchies

Sometimes you'd be tempted to create deeply nested inheritance hierarchies as you might be used to do in Java. For example, let's assume we're tempted to model something like that:

```cds
abstract entity Grantees { // equivalent to aspect
  key name : String;
}
entity Users : Grantees {
  group : Association to Groups;
}
entity Groups : Grantees {
  members : Composition of many Users on members.group = $self;
}
```

When combining that with relational persistence, you'll always end up in trade-off decisions about which strategy to choose for mapping such class hierarchies to flat tables. As that choice heavily depends on the use cases, CDS intentionally doesn't provide any automatic mapping of such inheritance hierarchies, but you have to choose one of the three commonly known approaches explicitly in your models as follows...

Avoid: Table Per Leaf Class Strategy

If we'd keep the model as given above, we'd end up with two separate tables, one for each leaf entity. The problem with that approach is that we'd need expensive UNIONs to, for example, display a heterogeneous list of Users and Groups. For example:

```cds
entity UsersAndGroups as (
  SELECT from Users
) UNION ALL (
  SELECT from Groups
);
```

Avoid: Table Per Class Strategy

If we want a separate table for each entity in our model above, including the 'superclass' entity `Grantees`, we'd have to rewrite our model to use composition over inheritance like that:

```cds
entity Grantees {
  key name : String;
}
entity Users {
  header : Association to Grantees;
  group : Association to Groups;
}
entity Groups {
  header : Association to Grantees;
  members : Composition of many Users on members.group = $self;
}
```

This would allow you to display heterogeneous lists of `Grantees` without UNIONs. A lot more JOINs would be required in real-world examples, though.

Prefer: Single Table Strategy

The third strategy is to put everything into a single table and an additional type discriminator element (→ *kind* in the sample below).

cds

```
entity Users {
  key name : String;
  kind : String enum { user; group }; // discriminator
  group : Association to Users;
  members : Composition of many Users on members.group = $self;
}
```

Advantages

- Simple model
- No UNIONs, no excess JOINs
- Bonus: deeply nested *Groups*

Previous page
Compiler Messages

Next page
The Nature of CDS Models