Корпоративный учет

Учет – это язык хозяйственной деятельности.

М.И. Куттер.

Сергей Минюров
Январь 2008 г.
Редакция 0.1 от 13 января 2008 г.

Содержание

1. Предметная модель[1]

Природа экономических явлений объясняется объективными законами естествознания, согласно которым ничего не возникает из ничего, материя переходит из одного состояния в другое. В экономике любой хозяйственный факт имеет два адреса: изменения одного объекта вызывают изменения на такую же величину другого объекта.

Предмет учета. Предметом бухгалтерского учета является производственно-хозяйственная и финансовая деятельность предприятия. Объекты учета различаются как обеспечивающие или составляющие деятельность.

Рисунок 1.1. Предмет учета и основные объекты бухгалтерского наблюдения

Объектами, обеспечивающими деятельность предприятия, являются *активы* и *пассивы*. Активы это хозяйственные средства (составляющие имущество), контролируемые организацией и приносящие ей экономические выгоды. Пассивы это источники хозяйственных средств – собственный капитал и заемные финансовые средства (обязательства).

Объектами, составляющими деятельность предприятия, являются *хозяйственные и финансовые процессы*, а также *финансовые результаты*.

Факты хозяйственной жизни. Процессы состоят из *фактов хозяйственной жизни* (ФХЖ), изменяющих финансовое положение экономического субъекта, либо оставляющих его неизменным. ФХЖ, оказывающие влияние на финансовое положение организации, называются *хозяйственными операциями*.

В бухгалтерском учете каждый факт хозяйственной жизни фиксируется в первичных документах и имеет три параметра идентификации:

- временной – момент регистрации операции в соответствии с учетной политикой предприятия;
- количественный - стоимостная оценка;
- квалификационный - классификация хозяйственной операции в номенклатуре плана счетов.

Бухгалтерский счет. Для ведения бухгалтерской информационной системы и отражения в ней состояния и движения каждого объекта, подлежащего бухгалтерскому наблюдению, применяется специальный регистр – *бухгалтерский счет.* На каждый вид актива, капитала и обязательств, а также на доходы и расходы открывается отдельный счет. Система бухгалтерских счетов называется *планом счетов*.

Бухгалтерские счета предназначены для отражения на них результатов воздействия фактов хозяйственной деятельности на объект, который учитывается на данном счете. Экономические воздействия имеют два направления: увеличение или уменьшение. Поэтому счет разбивается на две

[1] Этот раздел написан на основе [6].

информационные зоны, каждая из которых предназначена для учета изменений, направленных на увеличение или уменьшение показателя, характеризующего состояние объекта. Исторически левая сторона счета называется *дебет*, а правая - *кредит*.

Значение показателя на начало отчетного периода называется *начальное сальдо*, а на конец – *конечное сальдо*.

Расположение начального сальдо в дебете или кредите для счетов активов и пассивов зависит от местоположения объекта в бухгалтерском балансе.

Счета активов и пассивов. Активы располагаются на левой стороне баланса, следовательно, начальное и конечное сальдо на счетах активов и увеличение показателя размещаются на левой стороне счета - по дебету.

Таблица 1.1. Структура счетов с дебетовым сальдо

Дебет	**Счета активов**	*Кредит*
Сальдо начальное (СНД)	Хозяйственные операции, приводящие к уменьшению, списанию или выбытию активов (-)	
Хозяйственные операции, отражающие увеличение активов (+)		
Дебетовый оборот (ДО)		Кредитовый оборот (КО)
Сальдо конечное (СКД)		

Пассивы располагаются на правой стороне баланса, а начальное и конечное сальдо на счетах пассивов и увеличение показателя размещаются на правой стороне счета – по кредиту.

Таблица 1.2. Структура счетов с кредитовым сальдо

Дебет	**Счета капитала и обязательств**	*Кредит*
Хозяйственные операции, отражающие уменьшение или расходование капитала, обязательств (-)		Сальдо начальное (СНК)
		Хозяйственные операции, приводящие к увеличению или формированию капитала, обязательств (+)
Дебетовый оборот (ДО)		Кредитовый оборот (КО)
		Сальдо конечное (СКК)

На активно-пассивных счетах учитываются два объекта: один относится к активам, другой – к пассивам. Соответственно, такие счета имеют два сальдо: по дебету и по кредиту.

Переменные счета. Отражение увеличения или уменьшения на бессальдовых (транзитных, переменных) счетах расходов и доходов, описывающих финансовые и хозяйственные процессы, зависит от их влияния на величину прибыли. Доходы увеличивают прибыль (капитал), соответственно, увеличение на счетах доходов находится на той же стороне, что и увеличение на счетах капитала – по кредиту. Расходы уменьшают прибыль, поэтому увеличение на счетах расходов располагается на стороне, противоположной увеличению на счете капитала – по дебету.

Синтетические счета, субсчета и аналитические счета. Бухгалтерская информационная система имеет горизонтальные и вертикальные связи. Горизонтальные связи – это корреспонденция бухгалтерских счетов, а вертикальные связи – это суммирование на уровне синтетических счетов значений субсчетов и аналитических счетов.

Обобщенные характеристики объектов (стоимостная оценка) отражаются на *синтетических счетах* (счета первого порядка). Для выделения групп объектов по свойствам и назначению определяются субсчета (счета второго порядка), как составной части синтетического счета.

Детализированные характеристики экземпляров объектов отражаются на *аналитических счетах* (счета третьего порядка). Учет, осуществляемый на синтетических и аналитических счетах, называется соответственно синтетическим и аналитическим.

Аналитический учет[1] помимо денежного показателя может иметь дополнительные натуральные измерения, соответствующие специфике объектов учета.

Балансовые и забалансовые счета. По уровню влияния на показатели финансовой отчетности (по отношению к бухгалтерскому балансу) бухгалтерские счета делятся на балансовые и забалансовые.

Имущество, принадлежащее или экономически контролируемое хозяйствующим субъектом, учитывается на балансовых счетах.

Арендованные основные средства, имущество, поступившее в некапитализируемый финансовый лизинг, товарно-материальные ценности, принятые на ответственное хранение или в переработку, а также материальные ценности, переданные другим экономическим субъектам и учитываемые в составе имущества этих субъектов, учитываются отдельно на забалансовых счетах.

[1] Аналитический учет в решениях на программной платформе ***Agile*** реализуется в логистике как уровень оперативного учета.

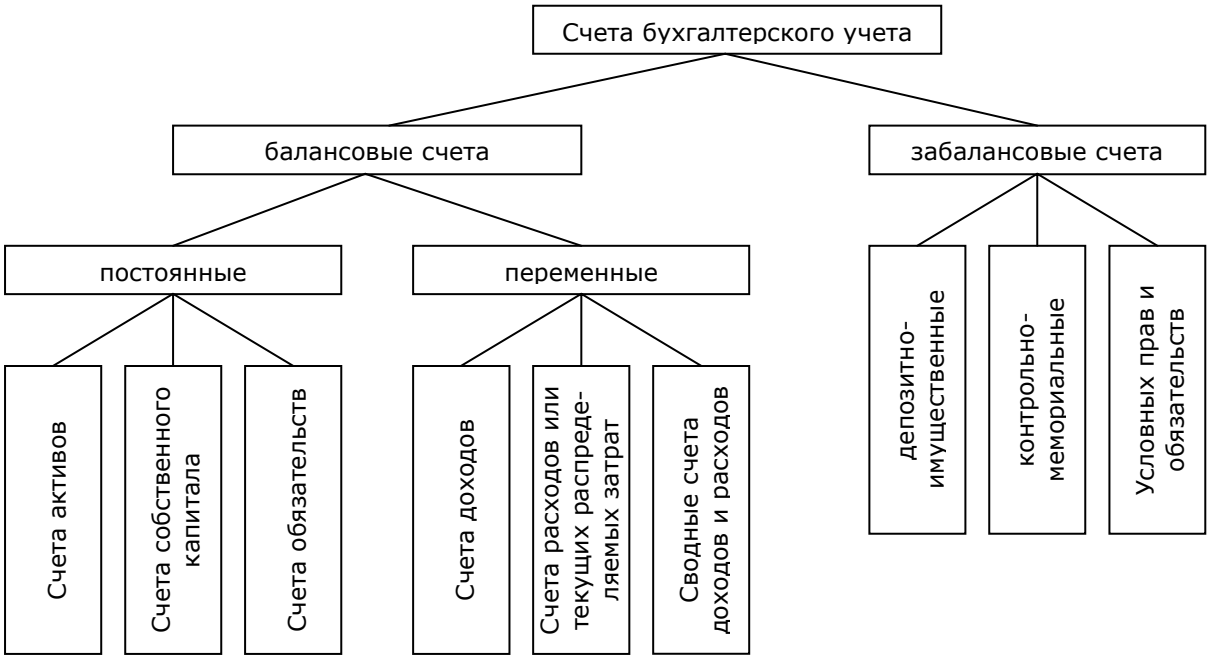


Рисунок 1.2. Классификация бухгалтерских счетов по уровню влияния на показатели финансовой отчетности (по отношению к бухгалтерскому балансу)

Хозяйственные операции. Каждый факт хозяйственной жизни описывается в первичных документах и регистрируется в абстрактной универсальной форме *хозяйственной операции* (двойной записи) - проведение по дебету и кредиту бухгалтерских счетов учета денежной суммы. Это является уравнением для связывания активов предприятия с их источниками, а также для связывания активов или источников средств при их преобразовании.

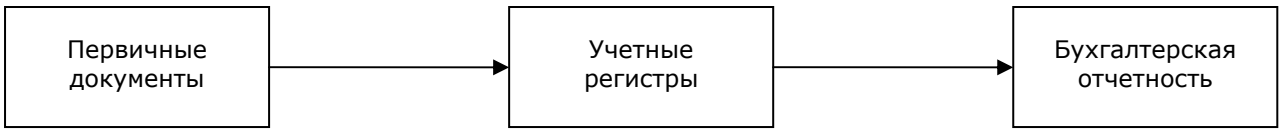


Рисунок 1.3. Движение информации о совершаемых ФХЖ

Типы хозяйственных операций:

1. Одновременное и равновеликое увеличение хозяйственных средств (активов) по дебету счета и их источников (капитала и обязательств) по кредиту счета. Валюта баланса увеличивается.
2. Изменение в составе имущества предприятия (активов): увеличение по дебету счета актива и уменьшение по кредиту другого счета актива. Валюта баланса не меняется.
3. Одновременное и равновеликое уменьшение в составе экономических ресурсов (активов) по кредиту счета активов и их источников (капитала и обязательств) по дебету счета пассивов. Валюта баланса уменьшается.
4. Изменение в составе источников средств предприятия (капитал и обязательства): увеличение по кредиту счета пассива и уменьшение по кредиту другого счета пассива. Валюта баланса не меняется.

1.1. Границы реализации

Бухгалтерский учет имеет аспекты определения схемы учета и регистрации учетных данных.

Схема учета описывается через понятия:

- *учетный счет*, *план счетов* и *раздел плана счетов*;
- *расчетный коэффициент* и его *периодическое значение* – коэффициент для расчета производных данных;
- *модель учета* – образец для формирования учетной операции;
- *тип учетного документа*, *форма учетного документа*, *форма отчета по документам*, *нумерация документов*.

Регистрация учетных данных описывается через понятия:

- *учетная операция* и *учетный документ*;
- *период учета* и *остатки за период*.

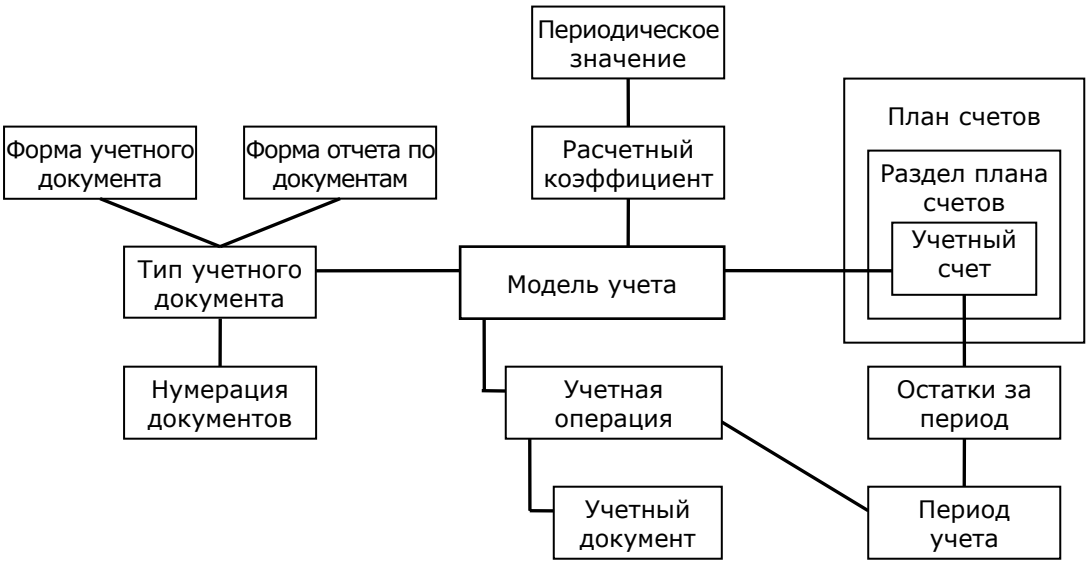


Рис. 1.4. Схема сущностей компонента «Корпоративный учет»

В учете может одновременно использоваться несколько планов счетов, например, основной план и план налогового учета. На основе плана счетов формируются остатки по учетным счетам за период. Поскольку в модели учета устанавливаются счета учета, то модель принадлежит конкретному плану счетов.

Учетная операция является первичным элементом учетных данных, регистрирующим факт хозяйственной жизни. Она формируется на основе модели учета, которая определяет корреспонденцию счетов и тип учетного документа.

> Модель учета и учетная операция могут иметь вложенные элементы, при этом для расчета сумм используются расчетные коэффициенты. Например, при формировании налоговой операции сумма определяется как процент от суммы первичной операции.
>
> Учетная операция реализована как производный класс от модели учета и имеет отношение наследования «один-ко-многим». Учетный документ реализован как производный класс от учетной операции.

Для корпоративного учета требуется возможность совместного ведения учета для нескольких хозяйствующих субъектов в разных валютах. Соответственно, для каждого плана счетов устанавливается базовая валюта и агент, для которого ведется учет. В учетной операции устанавливается текущая валюта и контрагент.

2. Схема учета

Схема учета определяется на двух уровнях:

- план счетов, состоящий из разделов и учетных счетов
- модель учета, которая является образцом для создания учетных операций

2.1. План счетов

Класс «План счетов» **Agile.Account.AccountPlan** наследуется от класса **Agile.Data.DataServiceContainer**.

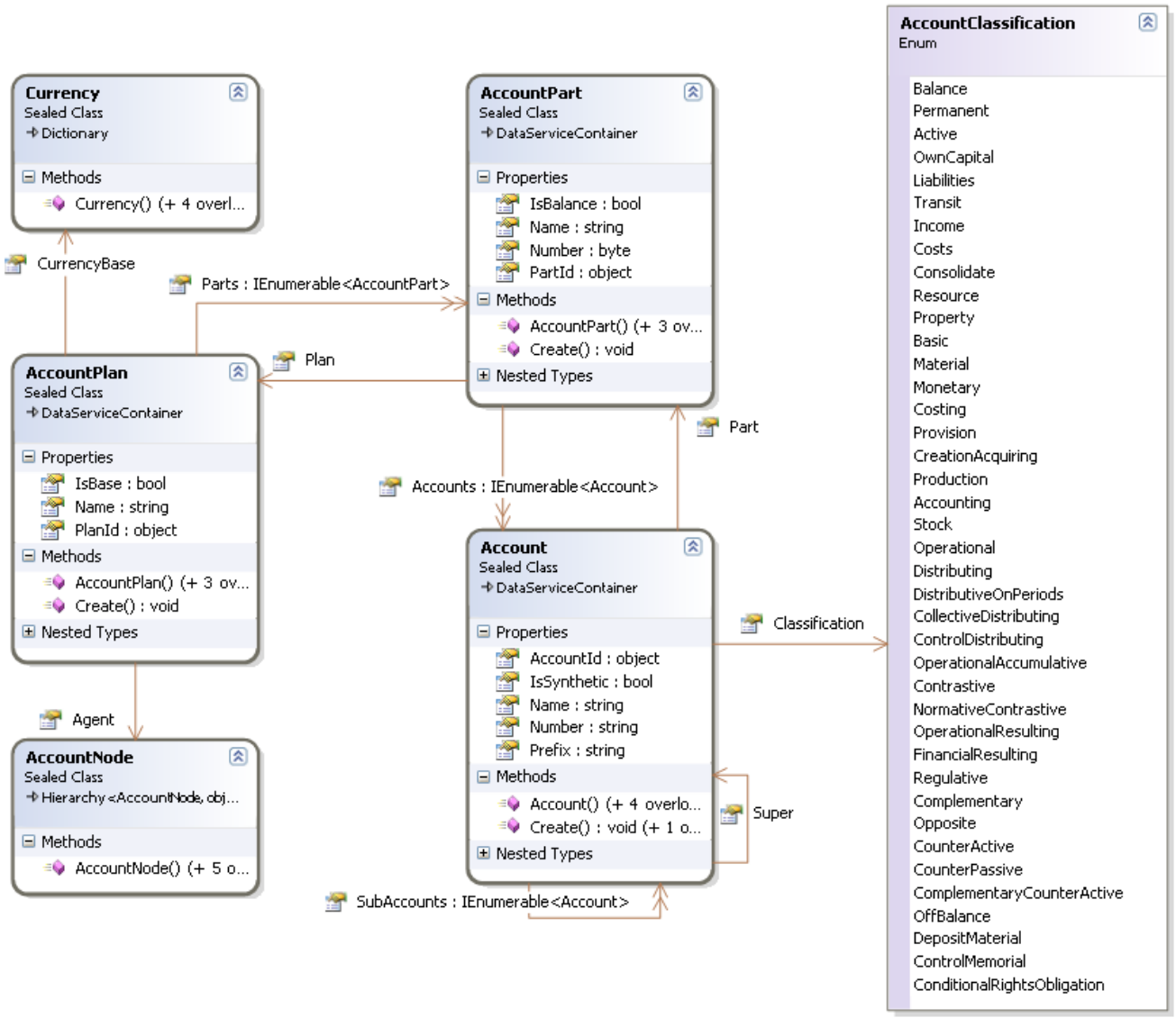


Рисунок 2.1. Диаграмма семейства классов плана счетов

В классе определено несколько конструкторов, в т.ч. конструктор по умолчанию. Конструктор с параметром `id` загружает экземпляр данных для объекта по его идентификатору. Конструктор с набором параметров, соответствующих свойствам объекта, создает новый экземпляр объекта.

Листинг 2.1. Конструкторы класса «План счетов»

```
// Конструктор по умолчанию
public AccountPlan()
// Конструктор для загрузки данных по идентификатору
public AccountPlan(object id)
// Конструктор для создания плана счетов
public AccountPlan(string name, AccountNode agent, Currency currency, bool isBase)
```

Таблица 2.1. Свойства класса «План счетов»

Название	Тип значения	Описание
PlanId	object	Идентификатор плана счета
Name	string	Название плана счетов
Agent	AccountNode	Агент плана счетов
IsBase	bool	Признак основного плана счетов для менеджера
CurrencyBase	Currency	Базовая валюта плана счетов
Parts	IEnumerable<AccountPart>	Список разделов плана счетов

Для создания нового экземпляра объекта определен метод **`Create`** с набором параметров. Этот метод вызывается в версиях конструктора для создания нового плана счетов.

Все параметры являются обязательными, при неопределенном значении любого из них генерируется программное исключение.

Листинг 2.2. Метод создания нового плана счетов

```
public void Create(string name, AccountNode agent, Currency currency, bool isBase)
```

2.1.1. Валюта [1]

Класс «Валюта» **Agile.Account.Currency** наследуется от справочника **Agile.Semantic.Dictionary** (см. [4]) и описывает список валют, используемых для определения учетных счетов.

В классе определено несколько конструкторов, в т.ч. конструктор по умолчанию. Конструктор с параметром `id` загружает экземпляр данных для объекта по его идентификатору. С параметром `sign` загружается экземпляр данных для объекта по обозначению.

Также определены конструкторы для создания нового узла. В соответствии со стандартным функционалом базового класса, если параметр `sign` определен, то проверяется существование данных с таким обозначением (см. [3]).

Листинг 2.3. Конструкторы класса «Валюта»

```
// Конструктор по умолчанию
public Currency()
// Конструктор для загрузки данных по идентификатору
public Currency(object id)
// Конструктор для загрузки данных по обозначению
public Currency(string sign)
// Конструктор для создания валюты
public Currency(string name, string shortName, string code, string sign)
```

Все свойства и методы наследуются от базового класса.

2.1.2. Учетный узел [2]

Класс «Учетный узел» **Agile.Account.AccountNode** наследуется от иерархического справочника **Agile.Semantic.Hierarchy<AccountNode, object>**.

Иерархический справочник позволяет определять справочное значение и устанавливать ссылку на другой объект, реализующий программные интерфейсы для контекстных сервисов и сервисов данных (см. [3]). Это позволяет интегрировать компонент учета с другими программными компонентами, разработанными независимо.

Например, можно создать учетный узел и установить в нем ссылку на предприятие как объект из внешнего компонента. А затем создать подчиненные узлы, также ссылающиеся на соответствующие объекты и описывающие его хозяйственную структуру в виде складов, магазинов, материально-ответственных лиц и т.д.

В классе определено несколько конструкторов, в т.ч. конструктор по умолчанию. Конструктор с параметром `id` загружает экземпляр данных для объекта по его идентификатору. С параметром `sign` загружается экземпляр данных для объекта по обозначению.

---

[1] Аналогичный класс Agile.Logistic.Currency определен в компоненте «Вычислительная модель логистики» `Agile.Logistic.dll`. Эти классы используют общую таблицу данных и имеют одинаковый глобальный идентификатор типа. Таким образом, обеспечивается независимость компонентов и интеграция на уровне данных.

[2] Аналогичный класс Agile.Logistic.ResourceNode определен в компоненте «Вычислительная модель логистики» `Agile.Logistic.dll`. Эти классы используют общую таблицу данных и имеют одинаковый глобальный идентификатор типа. Таким образом, обеспечивается независимость компонентов и интеграция на уровне данных.

Также определены конструкторы для создания нового узла. В соответствии со стандартным функционалом базового класса, если параметр `sign` определен, то проверяется существование данных с таким обозначением (см. [3]).

Листинг 2.4. Конструкторы класса «Учетный узел»

```
// Конструктор по умолчанию
public AccountNode()
// Конструктор для загрузки данных по идентификатору
public AccountNode(object id)
// Конструктор для загрузки данных по обозначению
public AccountNode(string sign)
// Конструкторы для создания учетного узла
public AccountNode(string name, AccountNode parent)
// Конструктор для создания нового подчиненного узла
public AccountNode(string name, string shortName, string code, string sign,
    AccountNode parent)
```

Все свойства и методы наследуются от базового класса.

2.2. Раздел плана счетов

Класс «Раздел плана счетов» **Agile.Account.AccountPart** наследуется от класса **Agile.Data.DataServiceContainer** и является частью плана счетов.

В классе определено несколько конструкторов, в т.ч. конструктор по умолчанию. Конструктор с параметром `id` загружает экземпляр данных для объекта по его идентификатору. Конструктор с набором параметров, соответствующих свойствам объекта, создает новый экземпляр объекта.

Листинг 2.5. Конструкторы «Раздел плана счетов»

```
// Конструктор по умолчанию
public AccountPart()
// Конструктор для загрузки данных по идентификатору
public AccountPart(object id)
// Конструктор для создания раздела плана счетов
public AccountPart(AccountPlan plan, byte number, string name, bool isBalance)
```

Таблица 2.2. Свойства класса «Раздел плана счетов»

Название	Тип значения	Описание
PartId	object	Идентификатор раздела плана счета
Number	byte	Номер раздела плана счетов
Name	string	Название раздела плана счетов
Plan	AccountPlan	План счетов
IsBalance	bool	Признак балансового раздела плана счетов
Accounts	IEnumerable<Account>	Список счетов раздела плана счетов

Для создания нового экземпляра объекта определен метод **`Create`** с набором параметров. Этот метод вызывается в версиях конструктора для создания нового раздела плана счетов.

Все параметры являются обязательными, при неопределенном значении любого из них генерируется программное исключение.

Листинг 2.6. Метод создания нового раздела плана счетов

```
public void Create(AccountPlan plan, byte number, string name, bool isBalance)
```

2.3. Учетный счет

Класс «Учетный счет» **Agile.Account.Account** наследуется от класса **Agile.Data.DataServiceContainer** и является частью раздела плана счетов.

В классе определено несколько конструкторов, в т.ч. конструктор по умолчанию. Конструктор с параметром `id` загружает экземпляр данных для объекта по его идентификатору. Конструктор с набором параметров, соответствующих свойствам объекта, создает новый экземпляр объекта.

Листинг 2.7. Конструкторы класса «Учетный счет»

```
// Конструктор по умолчанию
public Account()
// Конструктор для загрузки данных по идентификатору
public Account(object id)
```

```
// Конструктор для создания учетного счета
public Account(AccountPart part, string number, string name, bool isSynthetic,
    AccountClassification classification)
// Конструктор для создания субсчета
public Account(Account super, string number, string name)
```

Таблица 2.3. Свойства класса «Учетный счет»

Название	Тип значения	Описание
AccountId	object	Идентификатор учетного счета
Number	string	Номер счета
Name	string	Наименование счета
Prefix	string	Префикс номера счета (для синтетических и аналитических счетов)
Part	AccountPart	Раздел плана учета
IsSynthetic	bool	Признак синтетического счета
Classification	AccountClassification	Классификация учетного счета
Super	Account	Вышестоящий счет (для синтетических и аналитических счетов)
SubAccounts	IEnumerable<Account>	Список подчиненных счетов (для синтетического счета)

Для создания нового экземпляра объекта определен метод **`Create`** с набором параметров. Этот метод вызывается в версиях конструктора для создания нового учетного счета.

Все параметры являются обязательными, при неопределенном значении любого из них генерируется программное исключение.

Версия метода с параметром `super` создает субсчет.

Если счет, указанный в параметре `super`, не является синтетическим (логическое свойство `IsSynthetic` имеет значение `false`), то генерируется программное исключение.

Листинг 2.8. Методы создания учетного счета

```
public void Create(AccountPart part, string number, string name, bool isSynthetic,
    AccountClassification classification)
// Создание субсчета
public void Create(Account super, string number, string name)
```

При создании учетного счета проверяется соответствие раздела плана счетов и классификации счета.

Если логический признак «Балансовый» `IsBalance` раздела плана счетов установлен как `true`, то счет, включаемый в данный раздел плана счетов может иметь набор значений кроме группы «Забалансовый» `OffBalance`.

И наоборот, если признак установлен как `false`, то счет может иметь набор значений только из группы «Забалансовый».

Перечисление «Классификация учетного счета» **Agile.Account.AccountDesignation** имеет следующие значения:

- `Balance` – Балансовый
  - `Permanent` – Постоянный (с сальдо)
    - `Active` – Активы
    - `OwnCapital` – Собственный капитал
    - `Liabilities` – Обязательства
  - `Transit` – Переменные
    - `Income` – Доходы
    - `Costs` – Расходы и затраты
    - `Consolidate` – Сводные
- `Resource` - Ресурсные
  - `Property` - Имущественные
    - `Basic` - Основные
      - `Material` - Материальные

        - `Monetary` - Монетарные
      - `Costing` - Калькуляционные
        - `Provision` - Заготовление
        - `CreationAcquiring` - Создание или приобретение
        - `Production` - Производственные
    - `Accounting` - Расчетные
    - `Stock` - Фондовые
  - `Operational` - Операционные
    - `Distributing` - Распределительные
      - `DistributiveOnPeriods` - Распределительные по периодам
      - `CollectiveDistributing` - Собирательно-распределительные
      - `ControlDistributing` - Контрольно-распределительные
    - `OperationalAccumulative` - Операционно-накопительные
    - `Contrastive` - Сопоставительные
      - `NormativeContrastive` - Нормативно-сопоставительные
      - `OperationalResulting` - Операционно-результатные
  - `FinancialResulting` - Финансово-результатные
  - `Regulative` – Регулирующие (уточняющие)
    - `Complementary` - Дополняющие
    - `Opposite` - Контрарные
      - `CounterActive` - Контрактивные
      - `CounterPassive` - Контрпассивные
    - `ComplementaryCounterActive` - Дополняющие-контрактивные
  - `OffBalance` - Забалансовые
    - `DepositMaterial` - Депозитно-имущественные
    - `ControlMemorial` - Контрольно-мемориальные
    - `ConditionalRightsObligation` - Условные права и обязательства

Листинг 2.9. Перечисление «Классификация учетного счета»

```
[Flags]
public enum AccountClassification : ulong
{
    Balance                         = 1,
    Permanent                       = 1 + 2,
    Active                          = 1 + 2 + 4,
    OwnCapital                      = 1 + 2 + 8,
    Liabilities                     = 1 + 2 + 16,
    Transit                         = 32,
    Income                          = 32 + 64,
    Costs                           = 32 + 128,
    Consolidate                     = 32 + 256,
    Resource                        = 512,
    Property                        = 512 + 1024,
    Basic                           = 512 + 1024 + 2048,
    Material                        = 512 + 1024 + 2048 + 4096,
    Monetary                        = 512 + 1024 + 2048 + 8192,
    Costing                         = 512 + 1024 + 16384,
    Provision                       = 512 + 1024 + 16384 + 32768,
    CreationAcquiring               = 512 + 1024 + 16384 + 65536,
    Production                      = 512 + 1024 + 16384 + 131072,
    Accounting                      = 512 + 262144,
```

```
    Stock                           = 512 + 524288,
    Operational                     = 1048576,
    Distributing                    = 1048576 + 2097152,
    DistributiveOnPeriods           = 1048576 + 2097152 + 4194304,
    CollectiveDistributing          = 1048576 + 2097152 + 8388608,
    ControlDistributing             = 1048576 + 2097152 + 16777216,
    OperationalAccumulative         = 1048576 + 33554432,
    Contrastive                     = 1048576 + 67108864,
    NormativeContrastive            = 1048576 + 67108864 + 134217728,
    OperationalResulting            = 1048576 + 67108864 + 268435456,
    FinancialResulting              = 1048576 + 536870912,
    Regulative                      = 1073741824,
    Complementary                   = 1073741824 + 2147483648,
    Opposite                        = 1073741824 + 4294967296,
    CounterActive                   = 1073741824 + 4294967296 + 8589934592,
    CounterPassive                  = 1073741824 + 4294967296 + 17179869184,
    ComplementaryCounterActive      = 1073741824 + 34359738368,
    OffBalance                      = 68719476736,
    DepositMaterial                 = 68719476736 + 137438953472,
    ControlMemorial                 = 68719476736 + 274877906944,
    ConditionalRightsObligation     = 68719476736 + 549755813888
}
```

2.4. Модель учета

Модель учета наследуется учетной операцией и учетным документом. Поэтому она реализована в двух классах.

Абстрактный класс «Базовая модель учета» **Agile.Account.AccountModelBase** наследуется от класса **Agile.Data.DataServiceContainer**. В нем определены общие свойства данных для производных классов. От него наследуются конкретный класс модели учета, а также базовый абстрактный класс учетной операции.

Таблица 2.4. Свойства класса «Базовая модель учета»

Название	Тип значения	Описание
Name	string	Наименование учетной модели
Sign	string	Обозначение учетной модели
Plan	AccountPlan	Учетный план
Debit	Account	Учетный счет по дебету
Credit	Account	Учетный счет по кредиту
OperationType	OperationTypes	Тип учетной операции
DocumentType	AccountDocumentType	Тип учетного документа

Перечисление «Типы хозяйственных операций» **Agile.Account.OperationTypes** имеет следующие значения:

- `Receipt` – поступление
- `Leave` – выбытие
- `Move` – перемещение
- `Change` - изменение

Листинг 2.10. Перечисление «Типы хозяйственных операций»

```
public enum OperationTypes : byte { Receipt = 1, Leave = 2, Move = 3, Change = 4 }
```

Класс «Модель учета» **Agile.Account.AccountModel** наследуется от **Agile.Account.AccountModelBase** и содержит дополнительные свойства и методы.

В классе определено несколько конструкторов, в т.ч. конструктор по умолчанию. Конструктор с параметром `id` загружает экземпляр данных для объекта по его идентификатору. Конструктор с набором параметров, соответствующих свойствам объекта, создает новый экземпляр объекта.

Листинг 2.11. Конструкторы класса «Модель учета»

```
// Конструктор по умолчанию
public AccountModel()
// Конструктор для загрузки данных по идентификатору
public AccountModel(object id)
```

```
// Конструктор для создания модели учета
public AccountModel(string name, string sign, Account debit, Account credit,
    OperationTypes operationType, AccountDocumentType docType)
// Конструктор для создания подчиненной модели учета
public AccountModel(AccountModel super, string name, Account debit,
    Account credit, DirectionOfCalculate calculate, AccountFactor factor)
```

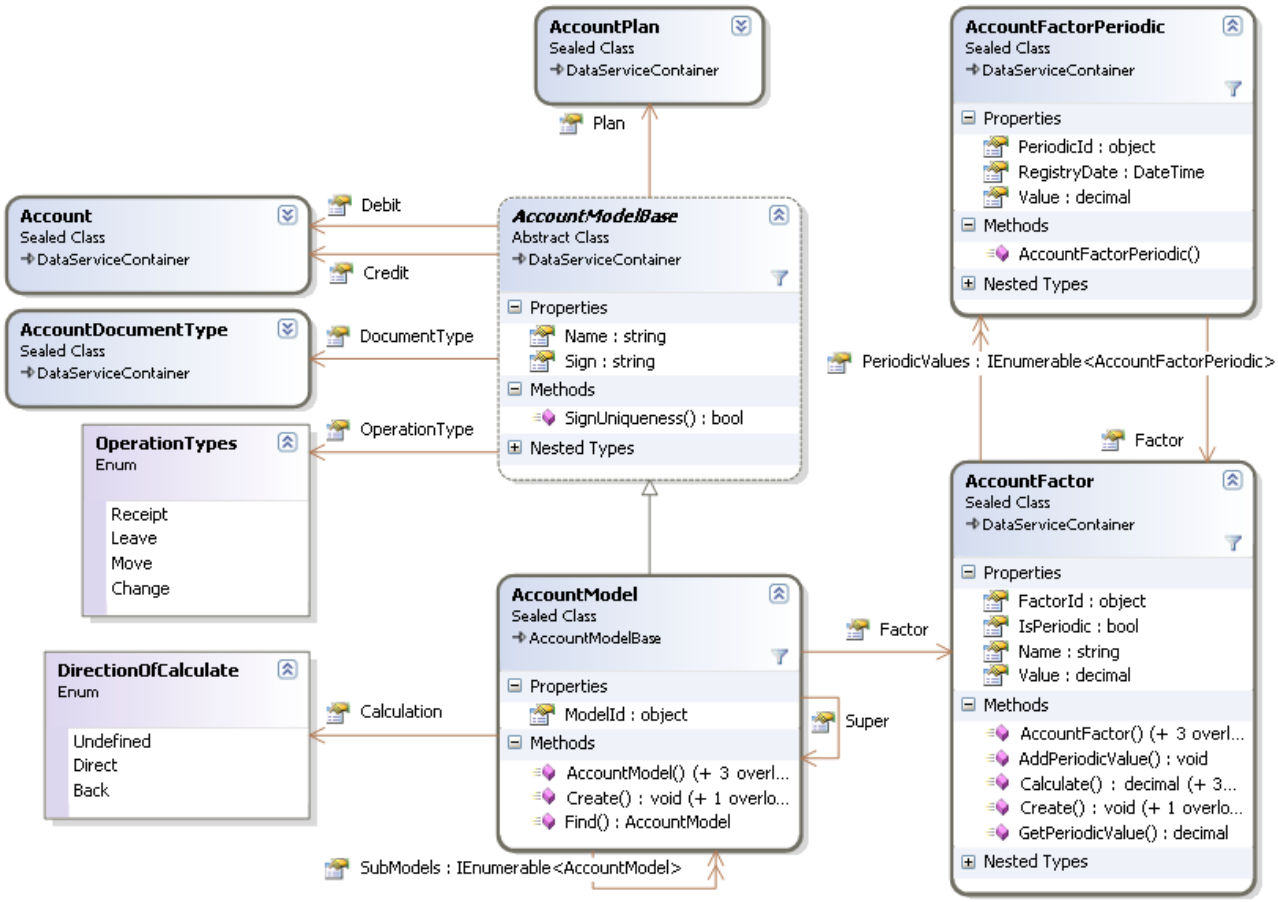


Рисунок 2.2. Диаграмма семейства классов модели учета

Таблица 2.5. Свойства класса «Модель учета»

Название	Тип значения	Описание
ModelId	object	Идентификатор модели учета
Factor	AccountFactor	Расчетный коэффициент
Calculation	DirectionOfCalculate	Направление расчета
Super	AccountModel	Вышестоящая модель учета
SubModels	IEnumerable<AccountModel>	Список подчиненных моделей учета

Для создания нового экземпляра объекта определен метод **`Create`** с набором параметров. Этот метод вызывается в версиях конструктора для создания новой модели учета.

Все параметры кроме параметра `docType` (тип учетного документа) являются обязательными, при неопределенном значении любого из них генерируется программное исключение.

Проверяется уникальность обозначения учетной операции. Если оно не уникально, то генерируется программное исключение.

Версия метода с параметром `super` создает подчиненную модель.

Все параметры кроме параметра `factor` (расчетный коэффициент) являются обязательными, при неопределенном значении любого из них генерируется программное исключение.

Если параметр `factor` не определен, то параметр `calculate` (направление расчета) должен иметь значение `Undefined`. Иначе, если параметр `factor` определен, то параметр `calculate` должен иметь значение `Direct` или `Back`. Соответственно, если параметр `calculate` имеет значение `Direct` или `Back`, то параметр `factor` должен быть определен. Иначе генерируется программное исключение.

Листинг 2.12. Методы создания модели учета

```
public void Create(string name, string sign, Account debit, Account credit,
    OperationTypes operationType, AccountDocumentType docType)
// Создание подчиненной модели учета
public void Create(AccountModel super, string name, Account debit, Account credit,
    DirectionOfCalculate calculate, AccountFactor factor)
```

Метод **SignUniqueness** проверяет уникальность обозначения модели учета. Если модель с данным обозначением существует, то возвращается значение `true`.

Листинг 2.13. Метод проверки уникальности обозначения модели учета

```
public bool SignUniqueness(string sign)
```

Перечисление «Направление расчета» **Agile.Account.DirectionOfCalculate** имеет следующие значения:

- `Undefined` – не определено
- `Direct` – прямой расчет
- `Back` – обратный расчет

Листинг 2.14. Перечисление «Направление расчета»

```
public enum DirectionOfCalculate : byte { Undefined = 0, Direct = 1, Back = 2 }
```

Статический метод **Find** производит поиск модели учета в соответствии с планом счетов и обозначением, установленных соответственно в параметрах `plan` и `sign`.

> Если данные не найдены и в параметре `throwIfNotFound` установлено значение `true`, то генерируется программное исключение **Agile.Data.DataNotFoundException**. Иначе возвращается неопределенное значение (`null`).

Листинг 2.15. Метод поиска модели учета по плану счетов и обозначению

```
public static AccountModel Find(AccountPlan plan, string sign,
    bool throwIfNotFound)
```

2.4.1. Расчетный коэффициент

Класс «Расчетный коэффициент» **Agile.Account.AccountFactor** наследуется от класса **Agile.Data.DataServiceContainer**. Расчетный коэффициент может иметь текущее или периодическое значение. Текущее значение хранится непосредственно таблицы данных самого объекта. Периодическое значение может иметь разное значение на разные даты и хранится в отдельной таблице данных.

В классе определено несколько конструкторов, в т.ч. конструктор по умолчанию. Конструктор с параметром `id` загружает экземпляр данных для объекта по его идентификатору. Конструктор с набором параметров, соответствующих свойствам объекта, создает новый экземпляр объекта.

Листинг 2.16. Конструкторы класса «Расчетный коэффициент»

```
// Конструктор по умолчанию
public AccountFactor()
// Конструктор для загрузки данных по идентификатору
public AccountFactor(object id)
// Конструктор для создания расчетного коэффициента
public AccountFactor(string name, decimal value, bool isPeriodic)
```

Таблица 2.6. Свойства класса «Расчетный коэффициент»

Название	Тип значения	Описание
FactorId	object	Идентификатор расчетного коэффициента
Name	string	Наименование расчетного коэффициента
Value	decimal	Значение расчетного коэффициента
IsPeriodic	bool	Признак периодического значения
PeriodicValues	IEnumerable<AccountFactorPeriodic>	Список периодических значений

Для создания нового экземпляра объекта определен метод **Create** с набором параметров. Этот метод вызывается в версиях конструктора для создания нового расчетного коэффициента.

Листинг 2.17. Метод создания расчетного коэффициента

```
public void Create(string name, decimal value, bool isPeriodic)
```

Текущее значение для непериодического расчетного коэффициента устанавливается через свойство **Value** (значение коэффициента). При этом если коэффициент является периодическим, то генерируется программное исключение.

Для добавления периодического значения определен метод **AddPeriodicValue**. Новое значение устанавливается как текущее для расчетного коэффициента. Дата регистрации, установленное в параметре `registryDate`, должна быть максимальной, иначе генерируется программное исключение.

Листинг 2.18. Метод добавления периодического значения

```
public AccountFactorPeriodic AddPeriodicValue(DateTime registryDate,
    decimal value)
```

Для получения периодического значения на дату определен метод **GetPeriodicValue**. Если значение не найдено, то возвращается нулевой результат.

Листинг 2.19. Метод получения периодического значения

```
public decimal GetPeriodicValue(DateTime registryDate)
```

Для расчета значения на основе коэффициента определен метод **Calculate**. Исходное значение передается в параметре `amount`.

Версия метода с одним только параметром amount производит расчет в прямом порядке по формуле: [исходное значение] * [значение коэффициента]. Например, при расчете налога НДС по ставке в 18% на исходную сумму в 1000 рублей получаем как результат сумму НДС в 180 рублей.

Версия метода с логическим параметром `back` позволяет рассчитать значение в прямом (значение `false`) или обратном порядке (значение `true`). Расчет в обратном порядке производится по формуле: ([исходное значение] / (1 + [значение коэффициента]) * [значение коэффициента]. Например, при общей сумме в 1000 рублей, в т.ч. с НДС 18%, получаем как результат сумму НДС в 152 рубля и 54 копейки.

Версии метода с параметром `registryDate` выполняет расчет для периодического коэффициента на указанную в этом параметре дату.

Листинг 2.20. Методы расчета значения

```
// Расчет значения для текущего значения коэффициента
public decimal Calculate(decimal sum)
public decimal Calculate(decimal sum, bool back)
// Расчет значения для периодического коэффициента
public decimal Calculate(decimal sum, DateTime registryDate)
public decimal Calculate(decimal sum, DateTime registryDate, bool back)
```

Класс «Периодическое значение коэффициента» **Agile.Account.AccountFactorPeriodic** наследуется от класса **Agile.Data.DataServiceContainer**.

В классе определен конструктор по умолчанию. Экземпляр класса создается в классе **Agile.Account.AccountFactor** при вызове метода **AddPeriodicValue**.

Листинг 2.21. Конструктор по умолчанию класса «Периодическое значение коэффициента»

```
public AccountFactorPeriodic()
```

Таблица 2.7. Свойства класса «Периодическое значение коэффициента»

Название	Тип значения	Описание
PeriodicId	object	Идентификатор периодического значения
Factor	AccountFactor	Расчетный коэффициент
RegistryDate	DateTime	Дата регистрации периодического значения
Value	decimal	Периодическое значение

3. Первичные данные по учету

Первичные учетные данные создаются в виде учетных операций на основе модели учета. Если в модели определен тип документа, то создается экземпляр учетного документа. Если модель является составной, т.е. в ней определены учетные подмодели, то в операции создаются подоперации.

От базовой модели учета наследуется базовая учетная операция, от которой затем наследуются учетная операция и учетный документ.

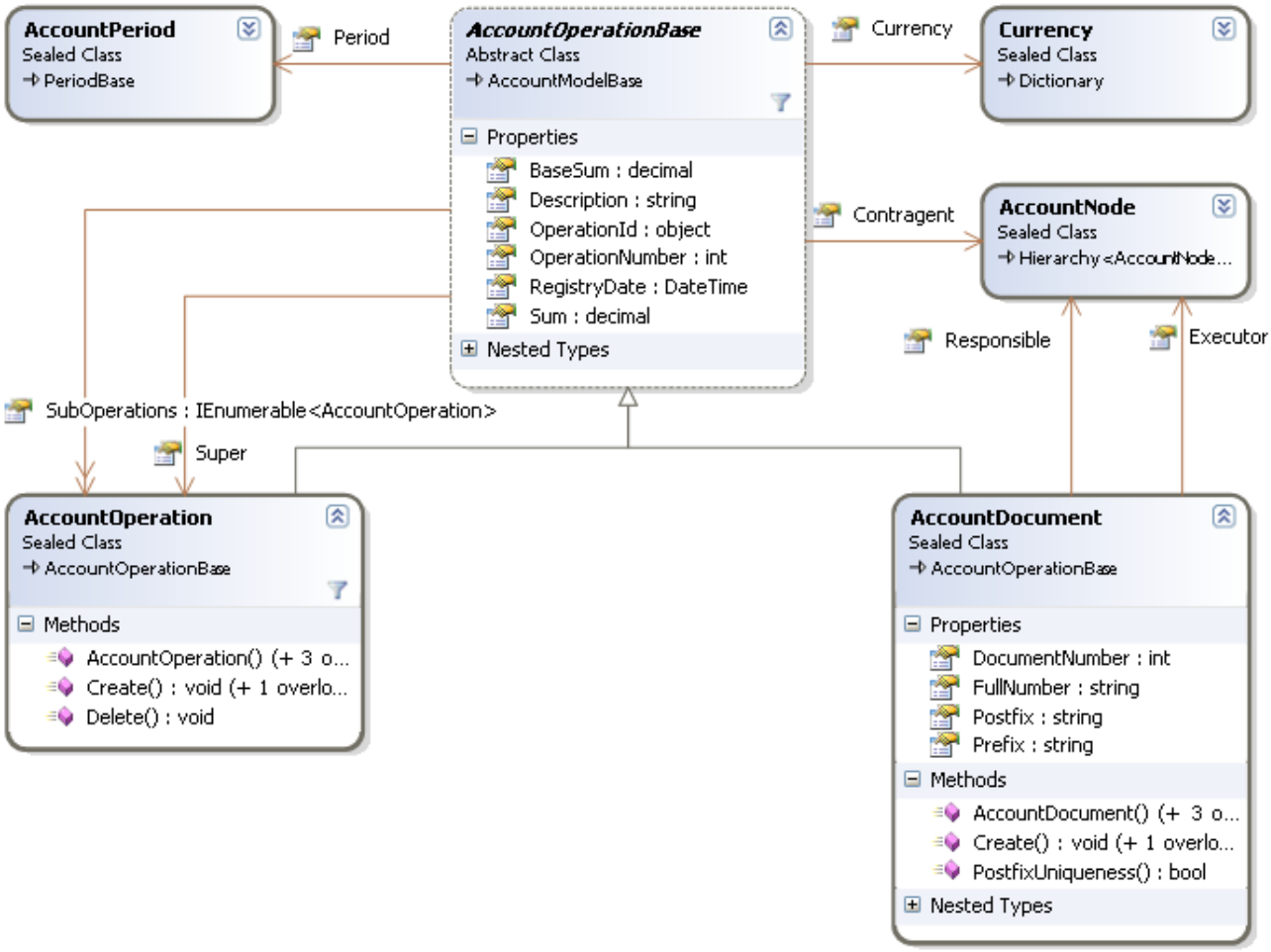


Рисунок 3.1. Диаграмма семейства классов первичных данных по учету

В учетной операции определено объектное свойство **`Contragent`**, которое ссылается на учетный узел, выполняющий роль *контрагент*. Агент определен в базовой модели учета как объектное свойство **`Agent`**.

В учетном документе определены объектные свойства **`Responsible`** и **`Executor`**, которые также ссылаются на учетные узлы, выполняющие роли *ответственный* и *исполнитель*.

3.1. Учетная операция

Абстрактный класс «Базовая учетная операция» **Agile.Account.AccountOperationBase** наследуется от **Agile.Account.AccountModelBase** на основе отношения «один-ко-многим» (на основе одной модели создается множество операций) и содержит свойства данных, общие для производных классов.

Таблица 3.1. Свойства класса «Базовая учетная операция»

Название	Тип значения	Описание
OperationId	object	Идентификатор учетной операции
OperationNumber	int	Порядковый номер операции
RegistryDate	DateTime	Дата регистрации операции
Period	AccountPeriod	Учетный период
Contragent	AccountNode	Контрагент операции
Currency	Currency	Валюта операции

Название	Тип значения	Описание
Sum	decimal	Сумма в валюте операции
BaseSum	decimal	Сумма в базовой операции
Description	string	Описание операции
Super	AccountOperation	Вышестоящая операция
SubOperations	IEnumerable<AccountOperation>	Список подчиненных операций

Класс «Учетная операция» **Agile.Account.AccountOperation** наследуется от **Agile.Account.AccountOperationBase** и содержит методы создания новой учетной операции.

В классе определено несколько конструкторов, в т.ч. конструктор по умолчанию. Конструктор с параметром `id` загружает экземпляр данных для объекта по его идентификатору. Конструктор с набором параметров, соответствующих свойствам объекта, создает новый экземпляр объекта.

Листинг 3.1. Конструкторы класса «Учетная операция»

```
// Конструктор по умолчанию
public AccountOperation()
// Конструктор для загрузки данных по идентификатору
public AccountOperation(object id)
// Конструктор для создания учетной операции
public AccountOperation(AccountModel model, AccountNode contragent,
    DateTime registryDate, decimal sum, string description)
// Конструктор для создания валютной учетной операции
public AccountOperation(AccountModel model, AccountNode contragent,
    DateTime registryDate, Currency currency, decimal sum, decimal baseSum,
    string description)
```

Для создания нового экземпляра объекта определен метод **`Create`** с набором параметров. Этот метод вызывается в версиях конструктора для создания новой учетной операции. Версия с дополнительными параметрами `currency` и `baseSum` создает валютную операцию.

Все параметры кроме `description` (описание операции) обязательные, при неопределенном значении любого из них генерируется программное исключение.

Порядковый номер операции устанавливается автоматически по максимальному значению с критерием отбора по плану счетов.

По дате регистрации операции, указанной в параметре `registryDate`, устанавливается связь с учетным периодом агента, который определен в плане счетов учетной модели.

Если период не определен, то генерируется программное исключение.

Для учетного периода устанавливается признак изменения, если период закрыт, то генерируется программное исключение.

Листинг 3.2. Методы создания учетной операции

```
public void Create(AccountModel model, AccountNode contragent,
    DateTime registryDate, decimal sum, string description)
// Создание валютной учетной операции
public void Create(AccountModel model, AccountNode contragent,
    DateTime registryDate, Currency currency, decimal sum, decimal baseSum,
    string description)
```

Создание учетной операции производится по учетной модели как образцу. Если в модели определен тип учетного документа, то дополнительно создается экземпляр учетного документа. Если модель составная, т.е. содержит подмодели, то, соответственно, создаются подчиненные операции.

3.2. Учетный документ

Класс «Учетный документ» **Agile.Account.AccountDocument** наследуется от класса **Agile.Account.AccountOperationBase**.

В классе определено несколько конструкторов, в т.ч. конструктор по умолчанию. Конструктор с параметром `id` загружает экземпляр данных для объекта по его идентификатору. Конструктор с набором параметров, соответствующих свойствам объекта, создает новый экземпляр объекта.

Листинг 3.3. Конструкторы класса «Учетный документ»

```
// Конструктор по умолчанию
public AccountDocument()
```

```
// Конструктор для загрузки данных по идентификатору
public AccountDocument(object id)
// Конструктор для создания учетного документа
public AccountDocument(AccountOperation operation, AccountNode responsible,
    AccountNode executor)
// Конструктор для создания учетного документа с номером существующего документа и
// постфиксом
public AccountDocument(AccountOperation operation, AccountNode responsible,
    AccountNode executor, AccountDocument document, string postfix)
```

Таблица 3.2. Свойства класса «Учетный документ»

Название	Тип значения	Описание
DocumentNumber	int	Порядковый номер документа
Prefix	string	Префикс номера документа
Postfix	string	Постфикс номера документа
FullNumber	string	Полный номер документа
Responsible	AccountNode	Ответственное лицо по документу
Executor	AccountNode	Исполнитель по документу

Свойство **`FullNumber`** формируется как последовательность, состоящая из префикса, порядкового номера и постфикса (если он есть). Разделители между ними определяются в параметрах компонента (см. далее раздел 6).

Для создания нового экземпляра объекта определен метод **`Create`** с набором параметров. Этот метод вызывается в версиях конструктора для создания нового учетного документа.

Версия с дополнительными параметрами `document` и `postfix` позволяет создать новый документ с тем же номером, но с уникальным постфиксом.

Если полный номер документа не уникальный, генерируется программное исключение.

Листинг 3.4. Методы создания учетного документа

```
public void Create(AccountOperation operation, AccountNode responsible,
    AccountNode executor)
// Создание учетного документа с номером существующего документа и постфиксом
public void Create(AccountOperation operation, AccountNode responsible,
    AccountNode executor, AccountDocument document, string postfix)
```

Метод **`PostfixUniqueness`** позволяет проверить уникальность полного номера документа с префиксом и номером документа, определенного в параметре `document`, и с постфиксом, указанным в параметре `postfix`. Этот метод используется методом **`Create`**.

Листинг 3.5. Метод проверки уникальности номера документа с постфиксом

```
public bool PostfixUniqueness(AccountDocument document, string postfix)
```

4. Учетный период и остатки

Бухгалтерский учет ведется в заданных временных рамках, которые определяются как учетные периоды[1]. Назначение учетных периодов заключается в фиксации итогов учета в виде остатков за период по счетам для оценки финансовых результатов.

В начале деятельности предприятия определяются начальные остатки, которые могут быть нулевыми. Затем конечные остатки предыдущего периода становятся начальными остатками следующего периода.

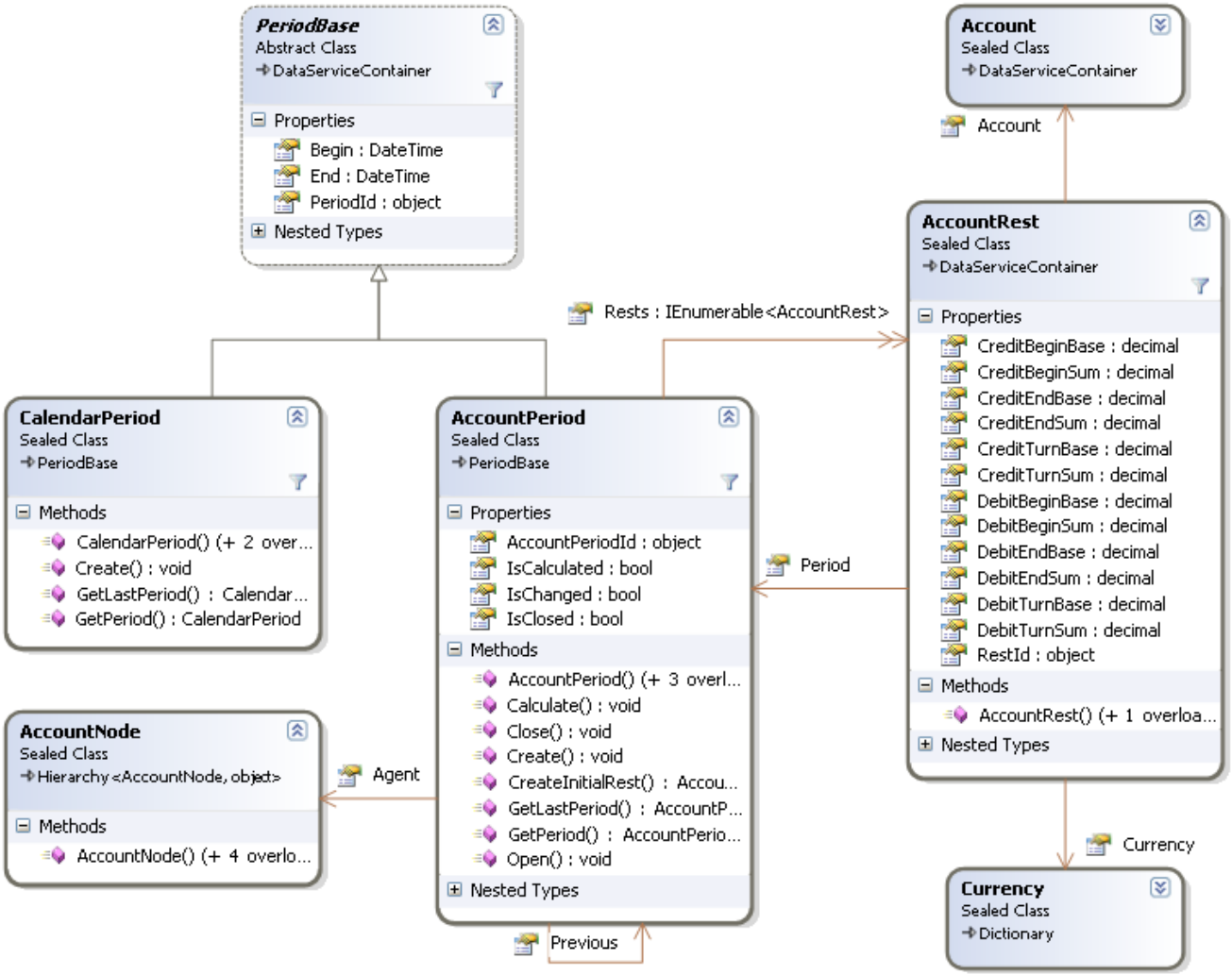


Рисунок 4.1. Диаграмма семейства классов периода учета и остатков

Для каждого агента, по которому ведется учет, отслеживается отдельное состояние по учетному периоду.

Остатки по учетному счету за период ведутся в разрезе валют и связаны с учетным периодом агента.

4.1. Период учета

Абстрактный класс «Базовый период» **Agile.Account.PeriodBase** наследуется от класса **Agile.Data.DataServiceContainer** и определяет свойства данных, общие для производных классов.

Таблица 4.1. Свойства класса «Базовый период»

Название	Тип значения	Описание
PeriodId	object	Идентификатор периода учета
Begin	DateTime	Дата начала учетного периода
End	DateTime	Дата окончания учетного периода

[1] Учетные периоды в бухгалтерском учете могут быть вложенными: месяц, квартал, полугодие, год. В компоненте реализованы функции только для ведения первичных учетных периодов.

Класс «Календарный период» **Agile.Account.CalendarPeriod** наследуется от класса **Agile.Account.PeriodBase** и определяет дополнительные методы.

В классе определено несколько конструкторов, в т.ч. конструктор по умолчанию. Конструктор с параметром `id` загружает экземпляр данных для объекта по его идентификатору. Конструктор с параметрами `begin` и `end` создает новый экземпляр объекта.

Листинг 4.1. Конструкторы класса «Календарный период»

```
// Конструктор по умолчанию
public CalendarPeriod()
// Конструктор для загрузки данных по идентификатору
public CalendarPeriod(object id)
// Конструктор для создания учетного периода
public CalendarPeriod(DateTime begin, DateTime end)
```

Для создания нового экземпляра объекта определен метод **`Create`** с набором параметров. Этот метод вызывается в версиях конструктора для создания нового учетного периода.

> Если временной диапазон пересекается с другим календарным периодом, либо не стыкуется с последним периодом, то генерируется программное исключение.

Листинг 4.2. Метод создания календарного периода

```
public void Create(DateTime begin, DateTime end)
```

Метод GetPeriod возвращает календарный период на дату, указанную в параметре `date`. Метод **`GetLastPeriod`** позволяет получить последний календарный период.

Листинг 4.3. Методы получения календарного периода на дату

```
// Получить календарный период на дату
public CalendarPeriod GetPeriod(DateTime date)
// Получить последний календарный период
public CalendarPeriod GetLastPeriod()
```

Класс «Учетный период» **Agile.Account.AccountPeriod** наследуется от класса **Agile.Account.PeriodBase**.

В классе определено несколько конструкторов, в т.ч. конструктор по умолчанию. Конструктор с параметром `id` загружает экземпляр данных для объекта по его идентификатору. Конструктор с параметрами `period` и `agent` создает новый экземпляр объекта.

Листинг 4.4. Конструкторы класса «Учетный период»

```
// Конструктор по умолчанию
public AccountPeriod()
// Конструктор для загрузки данных по идентификатору
public AccountPeriod(object id)
// Конструктор для создания периода учета агента
public AccountPeriod(CalendarPeriod period, AccountNode agent)
```

Таблица 4.2. Свойства класса «Учетный период»

Название	Тип значения	Описание
AccountPeriodId	object	Идентификатор периода учета агента
Agent	AccountNode	Агент
IsChanged	bool	Признак измененных данных за период учета
IsCalculated	bool	Признак расчета данных за период учета
IsClosed	bool	Признак закрытого периода учета
Previous	AccountPeriod	Предыдущий период учета агента
Rests	IEnumerable<AccountRest>	Остатки по учетным счетам за период

У нового периода все признаки (**`IsChanged`**, **`IsCalculated`** и **`IsClosed`**) первоначально имеют значение `false`. При вводе данных по учетным операциям устанавливается значение `true` для признака изменения данных за период **`IsChanged`**.

Установка значения `true` для признака закрытого периода учета **`IsClosed`** блокирует ввод и изменение данных по учетным операциям за этот период - при попытке генерируется программное исключение.

> Если устанавливается значение `true` для свойства **`IsClosed`**, и при этом предыдущий период не закрыт, то генерируется программное исключение.
>
> Если устанавливается значение `false` для свойства **`IsClosed`**, и при этом определен последующий период, который закрыт, то генерируется программное исключение.

Для создания нового экземпляра объекта определен метод **Create** с набором параметров. Этот метод вызывается в версиях конструктора для создания нового учетного периода.

При создании нового периода определяется текущий период и устанавливается на него ссылка как на предыдущий (свойство **Previous**). Первый период будет иметь неопределенную ссылку на предыдущий период и для него можно ввести начальные остатки по счетам.

Если заданный период учета уже определен для агента, то генерируется программное исключение.

Листинг 4.5. Метод создания учетного периода

```
public void Create(CalendarPeriod period, AccountNode agent)
```

Для ввода начальных остатков определен метод **CreateInitialRest**.

Начальные остатки можно вводить, только если период является первым, т.е. имеет неопределенное значение (null) для свойства **Previous**. Иначе генерируется программное исключение.

Определены версии метода для ввода остатков в базовой валюте или в валюте учета.

Если начальный остаток по учетному счету и валюте уже установлен, генерируется программное исключение.

Листинг 4.6. Методы установки начального остатка по учетному счету за период

```
// Установка начального остатка в базовой валюте
public AccountRest CreateInitialRest(Account account, decimal debitSum,
    decimal creditSum)
// Установка валютного начального остатка
public AccountRest CreateInitialRest(Account account, Currency currency,
    decimal debitSum, decimal debitBase, decimal creditSum, decimal creditBase)
```

Для расчета остатков за период определен метод **Calculate**, который рассчитывает обороты и конечные остатки по счетам. После успешного завершения этого метода устанавливается значение true для признака расчета данных за период **IsCalculated** и значение false для признака изменения данных за период **IsChanged**.

Если определен предыдущий период, по которому не выполнен расчет, то генерируется программное исключение.

Если период закрыт, то генерируется программное исключение.

Листинг 4.7. Метод расчета данных по остаткам по учетным счетам за период

```
public void Calculate()
```

Метод **Close** устанавливает период в закрытое состояние (значение true свойства **IsClosed**), в котором блокируется изменение учетных данных за этот период.

Если остатки за период не рассчитаны, то генерируется программное исключение.

При попытке создания учетной операции за закрытый период или изменения генерируется программное исключение.

Листинг 4.8. Метод закрытия учетного периода

```
public void Close()
```

Метод **Open** устанавливает период в открытое состояние (значение false свойства **IsClosed**), в котором разрешаются изменения учетных период за этот период.

Если определен следующий учетный период, который закрыт, то генерируется программное исключение, и период остается в закрытом состоянии.

Листинг 4.9. Метод открытия учетного периода

```
public void Open()
```

Для получения последнего учетного периода определен статический метод **GetLastPeriod**, которому в параметре agent передается агент, для которого требуется найти результат. Если агент не имеет учетных периодов, то возвращается неопределенный результат (null).

Листинг 4.10. Метод получения последнего периода учета агента

```
public static AccountPeriod GetLastPeriod(AccountNode agent)
```

Также определен статический метод **GetPeriod** для получения периода на дату, указанную в параметре date.

Версия с двумя параметрами `agent` и `date` и возвращает `null`, если период не найден.

Версия с дополнительным логическим параметром `throwIfUndefined`, установленного в `true` генерирует программное исключение, если период не найден.

Листинг 4.11. Методы получения периода учета агента на дату

```
public static AccountPeriod GetPeriod(AccountNode agent, DateTime date)
public static AccountPeriod GetPeriod(AccountNode agent, DateTime date,
    bool throwIfUndefined)
```

4.2. Остатки по учетному счету

Класс «Остатки по учетному счету» **Agile.Account.AccountRest** наследуется от класса **Agile.Data.DataServiceContainer** и содержит остатки по учетному счету за учетный период агента в разрезе валют.

В классе определено несколько конструкторов, в т.ч. конструктор по умолчанию. Конструктор с параметром `id` загружает экземпляр данных для объекта по его идентификатору.

Листинг 4.12. Конструкторы класса «Остатки по учетному счету»

```
// Конструктор по умолчанию
public AccountRest()
// Конструктор для загрузки данных по идентификатору
public AccountRest(object id)
```

Экземпляры типа формируются при выполнении метода **`Calculate`** класса «Учетный период» **Agile.Account.AccountPeriod** (см. предыдущий раздел 4.1).

Таблица 4.3. Свойства класса «Остатки по учетному счету»

Название	Тип значения	Описание
RestId	object	Идентификатор остатков за период по учетному счету
Account	Account	Учетный счет
Currency	Currency	Валюта учета
Period	AccountPeriod	Учетный период агента
DebitBeginSum	decimal	Начальный остаток по дебету в валюте учета
DebitBeginBase	decimal	Начальный остаток по дебету в базовой валюте плана счетов
DebitTurnSum	decimal	Оборот по дебету в валюте учета
DebitTurnBase	decimal	Оборот по дебету в базовой валюте плана счетов
DebitEndSum	decimal	Конечный остаток по дебету в валюте учета
DebitEndBase	decimal	Конечный остаток по дебету в базовой валюте плана счетов
CreditBeginSum	decimal	Начальный остаток по кредиту в валюте учета
CreditBeginBase	decimal	Начальный остаток по кредиту в базовой валюте плана счетов
CreditTurnSum	decimal	Оборот по кредиту в валюте учета
CreditTurnBase	decimal	Оборот по кредиту в базовой валюте плана счетов
CreditEndSum	decimal	Конечный остаток по кредиту в валюте учета
CreditEndBase	decimal	Конечный остаток по кредиту в базовой валюте плана счетов

Остатки рассчитываются по дебету и кредиту, имеют значения на начало периода, оборот за период, и на окончание периода. Конечные остатки предыдущего периода равны начальным остаткам последующего периода.

Сумма рассчитывается в учетной и базовой валюте. Соответственно, если учетная и базовая валюты совпадают, то и эти суммы будут равны между собой.

5. Документы

Учетные документы создаются на основе учетных операций. Определены типы документов, в которых задается нумерация, формы и отчеты документов.

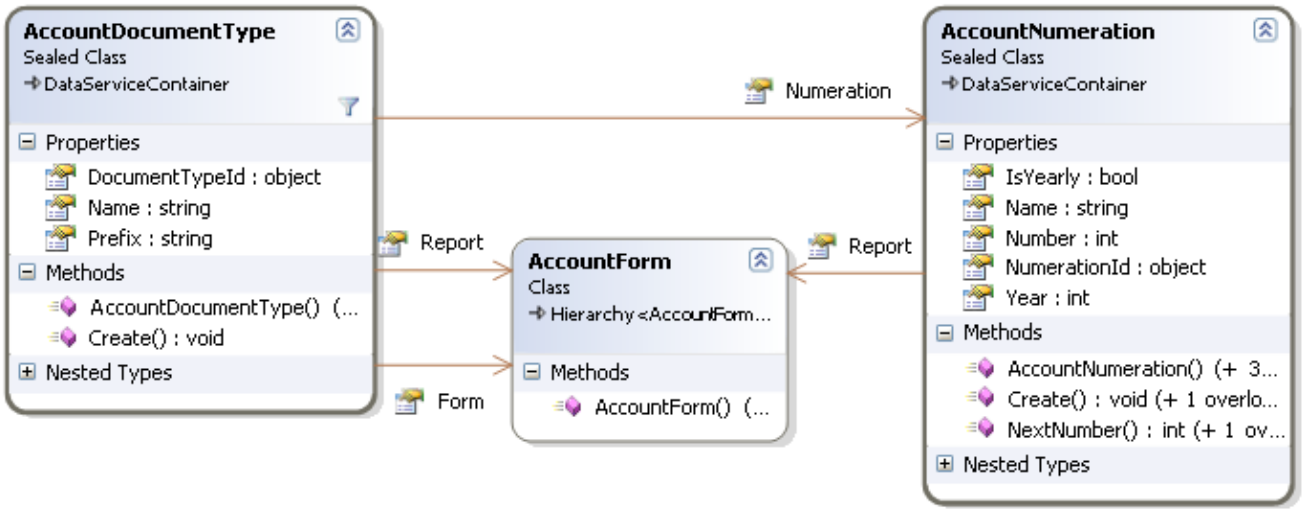


Рисунок 5.1. Диаграмма семейства классов типов документов

5.1. Тип учетного документа

Класс «Тип учетного документа» **Agile.Account.AccountDocumentType** наследуется от класса **Agile.Data.DataServiceContainer**, является справочником видов учетных документов.

В классе определено несколько конструкторов, в т.ч. конструктор по умолчанию. Конструктор с параметром `id` загружает экземпляр данных для объекта по его идентификатору. Конструктор с набором параметров, соответствующих свойствам объекта, создает новый экземпляр объекта.

Листинг 5.1. Конструкторы класса «Тип учетного документа»

```
// Конструктор по умолчанию
public AccountDocumentType()
// Конструктор для загрузки данных по идентификатору
public AccountDocumentType(object id)
// Конструктор для создания типа учетного документа
public AccountDocumentType(string name, AccountForm form, AccountForm report,
    AccountNumeration numeration, string prefix)
```

Таблица 5.1. Свойства класса «Тип учетного документа»

Название	Тип значения	Описание
DocumentTypeId	object	Идентификатор типа документа
Name	string	Наименование типа документа
Numeration	AccountNumeration	Нумерация документа
Prefix	string	Префикс номера документа
Form	AccountForm	Форма документа
Report	AccountForm	Отчет по документу

Для создания нового экземпляра объекта определен метод **`Create`** с набором параметров. Этот метод вызывается в версиях конструктора для создания нового типа учетного документа.

Листинг 5.2. Метод создания типа учетного документа

```
public void Create(string name, AccountForm form, AccountForm report,
    AccountNumeration numeration, string prefix)
```

5.2. Форма учетного документа

Класс «Форма учетного документа» **Agile.Account.AccountForm** наследуется от иерархического справочника **Agile.Semantic.Hierarchy< AccountForm, object >** (см. [4]) и описывает формы представления для печати документа.

В классе определено несколько конструкторов, в т.ч. конструктор по умолчанию. Конструктор с параметром `id` загружает экземпляр данных для объекта по его идентификатору. С параметром `sign` загружается экземпляр данных для объекта по обозначению.

Также определены конструкторы для создания новой формы документа. В соответствии со стандартным функционалом базового класса, если параметр `sign` определен, то проверяется существование данных с таким обозначением (см. [3]).

Листинг 5.3. Конструкторы справочника «Форма учетного документа»

```
// Конструктор по умолчанию
public AccountForm()
// Конструктор для загрузки данных по идентификатору
public AccountForm(object id)
// Конструктор для загрузки данных по обозначению
public AccountForm(string sign)
// Конструктор для создания формы
public AccountForm(string name, string shortName, string code, string sign)
// Конструктор для создания подчиненной формы
public AccountForm(string name, string shortName, string code, string sign,
    AccountForm parent)
```

Все свойства и методы наследуются от базового класса.

5.3. Нумерация документов

Класс «Нумерация документов» **Agile.Account.AccountNumeration** наследуется от класса **Agile.Data.DataServiceContainer** и содержит счетчики для нумерации учетных документов.

В классе определено несколько конструкторов, в т.ч. конструктор по умолчанию. Конструктор с параметром `id` загружает экземпляр данных для объекта по его идентификатору. Конструктор с набором параметров, соответствующих свойствам объекта, создает новый экземпляр объекта.

Листинг 5.4. Конструкторы класса «Нумерация документов»

```
// Конструктор по умолчанию
public AccountNumeration()
// Конструктор для загрузки данных по идентификатору
public AccountNumeration(object id)
// Конструктор для создания новой постоянной нумерации
public AccountNumeration(string name, int number, AccountForm report)
// Конструктор для создания новой ежегодной нумерации
public AccountNumeration(string name, int number, AccountForm report, int year)
```

Таблица 5.2. Свойства класса «Нумерация документов»

Название	Тип значения	Описание
NumerationId	object	Идентификатор нумерации документов
Name	string	Наименование нумерации документов
Number	int	Текущий номер
IsYearly	bool	Признак ежегодной нумерации
Year	short	Текущий год нумерации (для ежегодной нумерации)
Report	AccountForm	Отчет по нумерации

Для создания нового экземпляра объекта определен метод **`Create`** с набором параметров. Этот метод вызывается в версиях конструктора для создания новой нумерации документов.

Листинг 5.5. Методы создания нумерации документов

```
// Создать постоянную нумерацию
public void Create(string name, int number, AccountForm report)
// Создать ежегодную нумерацию
public void Create(string name, int number, AccountForm report, int year)
```

Для получения следующего номера определен метод **`NextNumber`**, который имеет версию с параметром `date` при ежегодной нумерации документов.

При запросе номера при ежегодной нумерации проверяется дата регистрации документа. Если текущий год нумерации меньше, чем год в значении параметра, то устанавливается начальное значение счетчика (1), и новый текущий год нумерации.

Листинг 5.6. Методы запроса следующего номера

```
// Запрос номера при постоянной нумерации
public int NextNumber()
// Запрос номера при ежегодной нумерации
public int NextNumber(DateTime registryDate)
```

6. Инфраструктура компонента

Инфраструктура компонента корпоративного учета состоит из классов контроллера компонента и параметров (см. [2]). При использовании компонента доступа к данным **Agile.Data** (см. [4]) как поставщика сервисов данных в xml-файле `Agile.Account.datamap` определяется схема данных.

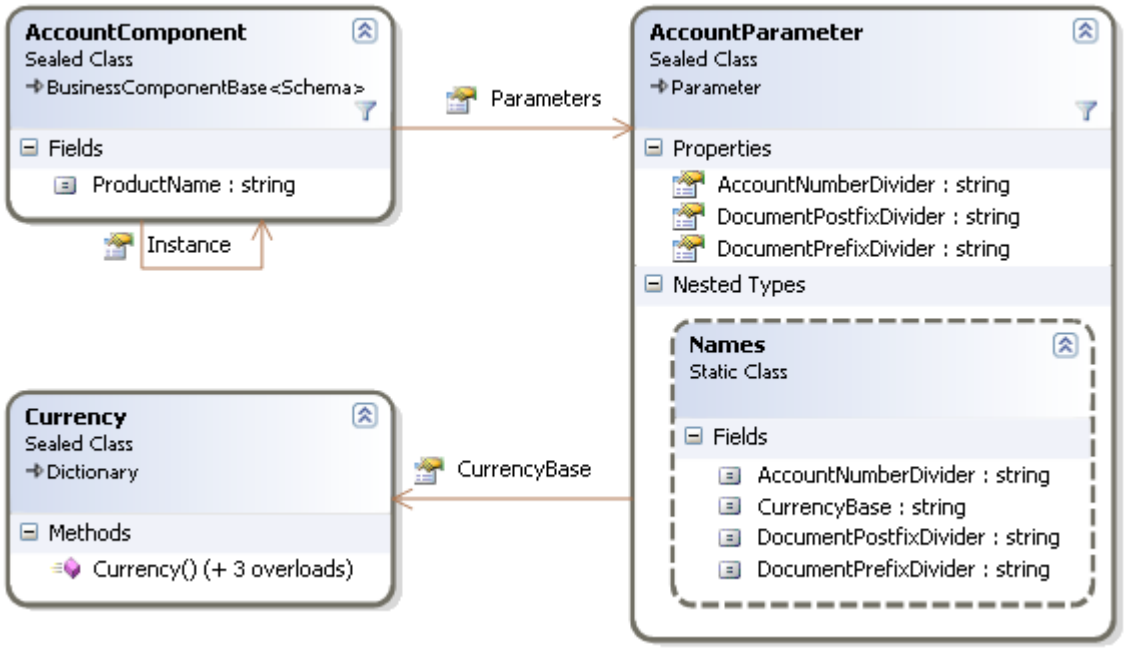


Рисунок 6.1. Диаграмма классов инфраструктуры компонентов

Класс «Компонент логистики» **Agile.Account.AccountComponent** наследуется от абстрактного generic-типа **Agile.Data.BusinessComponentBase<Agile.Account.Data.Schema>** и является контроллером компонента, в который устанавливается сервис источника данных и фабрика сервисов данных для классов компонента.

В статическом поле **`ProductName`** определяется системное название компонента.

В статическом свойстве **`Instance`** для классов компонента предоставляется доступ к экземпляру контроллера. При этом сам экземпляр запрашивается у менеджера компонентов (класс **Agile.Components.ComponentManager**, см. [4]).

Листинг 6.1. Статическое свойство экземпляра контроллера компонента

```
public static AccountComponent Instance {
    get { return ComponentManager.GetComponent<AccountComponent>(); }
}
```

Тип контроллера компонента определяется в атрибуте сборки **Agile.Components.ManagedComponentAttribute**. При динамической загрузке компонентов по этому атрибуту определяются контроллеры для управления этими компонентами.

Листинг 6.2. Определение типа контроллера компонента в AssemblyInfo.cs

```
[assembly: ManagedComponent(typeof(AccountComponent))]
```

Класс «Параметры учета» **Agile.Account.AccountParameters** наследуется от класса **Ãgile.Semantic.ParameterSet** (см. [3]) и содержит хранимые в базе данных пользовательские настройки.

Таблица 6.1. Свойства класса «Параметры учета»

Название	Тип значения	Описание
CurrencyBase	Currency	Базовая валюта корпоративная учета
AccountNumberDivider	string	Разделитель номера учетного счета
DocumentPrefixDivider	string	Разделитель префикса и номера учетного документа
DocumentPostfixDivider	string	Разделитель номера и постфикса учетного документа

7. Модель программирования

Модель программирования для компонента корпоративного учета описывает порядок действий для формирования схемы учета, ведения первичного учета и интеграции с другими бизнес-компонентами.

Типовые задачи по ведению бухгалтерского учета:

1. Определение плана счетов
2. Определение типов документов
3. Определение модели учета
4. Определение периода учета и начальных остатков
5. Создание учетных операций и расчет остатков за период

Пример интеграции сделан для компонента «Вычислительная модель логистики», и описывает один из подходов.

7.1. Определение плана счетов

В корпоративном учете в одной системе может вестись мультивалютный учет одновременно для нескольких предприятий, каждое из которых может иметь несколько схем учета, которым соответствуют отдельные планы счетов и модели учета.

Для создания плана счетов требуется определить базовую валюту учета и агента (хозяйствующего субъекта), для которого ведется учет.

Агент определяется как экземпляр учетного узла, для которого может быть установлена интеграция с бизнес-объектом из другого компонента.

Листинг 7.1. Создание плана счетов

```
// Определение учетного узла по обозначению, являющегося агентом, для которого
// создается план счетов
AccountNode myEnterprise = new AccountNode("MyEnterprise");
// Определение базовой валюты плана счетов по обозначению
Currency rub = new Currency("RUB");
// Создание основного плана счетов для агента
AccountPlan basePlan = new AccountPlan("Основной план счетов", myEnterprise,
    rub, true);
```

В плане счетов создаются разделы счетов, для которых устанавливается порядковый номер и признак балансовых счетов.

Листинг 7.2. Создание раздела плана счетов

```
// Создание раздела плана счетов
AccountPart part1 = new AccountPart(basePlan, 1, "Внеоборотные активы", true);
```

Учетный счет привязывается к разделу счетов и имеет уникальный код, название, логический признак синтетического счета и классификационные признаки.

Синтетический счет не может быть установлен в модели учета, но по нему формируются остатки по учету как сводный результат по его субсчетам.

Листинг 7.3. Создание учетного счета

```
// Создание синтетического учетного счета
Account acc01 = new Account(part1, "01", "Основные средства", true,
    AccountClassification.Active | AccountClassification.Material);
// Создание подчиненного учетного счета
Account acc01_1 = new Account(acc01, "1", "Сырье и материалы");
```

7.2. Определение типов документов

Для определения типов документов необходимо предварительно создать первичные и отчетные формы документов, а также нумерацию документов. Эти объекты являются общими для схем учета разных агентов.

Форма документов имеет полное и краткое наименование, код и обозначение.

Листинг 7.4. Создание форм документов

```
AccountForm debitCashForm = new AccountForm("Приходный кассовый ордер", "ПКО",
    "КО-1", "DebitCashWarrant");
AccountForm cashReportForm = new AccountForm("Кассовый отчет", "КО", "5-Г",
    "CashReport");
```

Нумерация документов имеет название, начальное значение, год для ежегодной нумерации и форма отчета по документам, которые связанны с данной нумерацией.

Листинг 7.5. Создание нумерации документов

```
AccountNumeration debitCashNumeration = new AccountNumeration(
    "Приходный кассовый документ", 0, 2007, cashReportForm);
```

Тип учетного документа имеет название, форму первичного документа и форму отчета по данному типу документа, нумерацию и префикс номеров, если он определен.

Листинг 7.6. Создание типа документа

```
AccountDocumentType debitCashType = new AccountDocumentType(
    "Приходный кассовый документ", debitCashForm, debitCashReport,
    debitCashNumeration, "");
```

7.3. Определение модели учета

Процесс учета по видам деятельности состоит из набора моделей учета, на основе которых последовательно создаются учетные операции.

> Модели учета связаны между собой через корреспонденцию счетов: счет, указанный в одной модели по дебету, в связанной модели будет указан по кредиту, и наоборот. При этом

В качестве примера рассмотрим учет реализации товаров за наличный расчет. На рисунке 7.1. и в таблице 7.1. описаны учетные операции, описывающие данный учетный процесс.

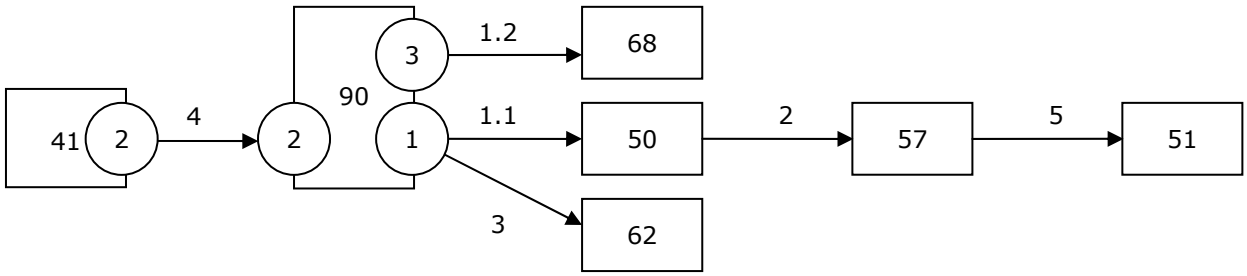


Рисунок 7.1. Порядок учета операций по реализации товаров за наличный расчет

Модели учета могут быть простыми – содержать только описание одной учетной операцией. Или составными – содержать описание нескольких учетных операций, в т.ч. с расчетом суммы подчиненной операции на основе заданного коэффициента.

> Например, первая модель по поступлению в кассу выручки за товар, имеет подчиненную модель по начислению НДС, в которой задан расчетный коэффициент по ставке налога для расчета суммы налога на основе суммы родительской операции. Остальные модели являются простыми и каждая описывает одну учетную операцию.

Таблица 7.1. Учетные операции по реализации товаров за наличный расчет

пп	Дебит	Кредит	Содержание операции
1	50	90.1	Поступление в кассу выручки за товар
	90.3	68	Начисление задолженности бюджету по НДС
2	57	50	Сдача выручки инкассаторам банка
3	62	90.1	Отражение выручки от продажи товара при их реализации «по отгрузке»
4	90.2	41.2	Списание реализованных товаров
5	51	57	Зачислена на расчетный счет выручка, находящаяся в пути

Ставка налога на добавленную стоимость определена как расчетный коэффициент, поскольку ее значение может меняться в соответствии с изменением законодательства, то данный коэффициент определен как периодический.

Листинг 7.7. Создание коэффициента для расчета НДС

```
// Создание периодического расчетного коэффициента
AccountFactor VATaxFactor = new AccountFactor("Ставка НДС");
```

```
// Добавление периодического значения
VATaxFactor.AddPeriodicValue(new DateTime(2004, 1, 1), 0.18);
```

Модель учета имеет название, обозначение, счет по дебету и кредиту, тип учетной операции и необязательное значение типа первичного учетного документа. Подчиненная модель не имеет обозначения и первичный документ, но может иметь расчетный коэффициент и порядок его вычисления.

Листинг 7.8. Создание составной модели учета

```
AccountModel receiptInCash = new AccountModel(
    "Поступление в кассу выручки за товар", "ReceiptInCashSaleProceeds", acc50,
    acc90_1, OperationTypes.Receipt, debitCashType);
AccountModel chargeVATax = new AccountModel(receiptInCash,
    "Начисление задолженности бюджету по НДС", acc90_3, acc68,
    DirectionOfCalculate.Back, VATaxFactor);
// Перечисление подчиненных моделей
foreach(AccountModel subModel in receiptInCash.SubModels)
    Assert.AreEquals(chargeVATax.ModelId, subModel.ModelId);
```

Листинг 7.9. Создание простых моделей учета

```
AccountModel collectMoney = new AccountModel("Сдача выручки инкассаторам банка",
    "CollectMoney", acc57, acc50, OperationTypes.Leave, creditCashType);
AccountModel saleProceeds = new AccountModel("Выручка от продажи товара",
    "SaleProceeds", acc62, acc90_1, OperationTypes.Change, null);
AccountModel goodsSoldDisc = new AccountModel("Списание реализованных товаров",
    "GoodsSoldDiscarding", acc90_2, acc41_2, OperationTypes.Leave, null);
AccountModel enterInAcc = new AccountModel("Выручка зачислена на расчетный счет",
    "EnterInAccountSaleProceeds", acc51, acc57, OperationTypes.Receipt,
    statementOfAccount);
```

7.4. Определение периода учета и начальных остатков

Каждый учетный счет, кроме бессальдовых счетов, имеет остаток за учетный период. При создании учетной операции устанавливается связь с учетным периодом для агента.

В корпоративном учете создается сначала календарный период, в котором устанавливаются даты его начала и окончания. Затем на основе календарного периода и агента создается учетный период для агента, который имеет логические признаки изменения данных, расчета остатков и закрытия периода.

> При создании учетного периода для агента определяется предыдущий период и устанавливается на него ссылка. Первый период учета имеет неопределенную ссылку на предыдущий период.

Листинг 7.10. Создание периода учета

```
// Создание календарного периода
CalendarPeriod january = new CalendarPeriod(new DateTime(2008, 1, 1),
    new DateTime(2008, 1, 31));
// Создание учетного периода для агента
AccountPeriod januaryMyEnt = new AccountPeriod(january, myEnterprise);
```

Для первого периода устанавливаются начальные остатки по счетам. Для мультивалютного счета отдельно устанавливаются остатки по каждому виду валюты.

Листинг 7.11. Установка начальных остатков за период

```
// Установка начального остатка в базовой валюте
AccountRest rest50 = januaryMyEnt.CreateInitialRest(acc50_1, 10000, 0);
// Установка начального остатка в валюте
Currency usd = new Currency("USD");
AccountRest rest50usd = januaryMyEnt.CreateInitialRest(acc50_1, usd, 1000, 25000,
    0, 0);
```

7.5. Создание учетных операций и расчет остатков за период

Учетная операция создается на основе учетной модели. Также для ее создания требуется указать контрагента, дату регистрации операции, сумму и описание. При создании мультивалютной операции нужно указать валюту операции. Агент и базовая валюта определяются из плана счетов, с которыми связаны счета по дебету и кредиту из модели учета.

Если учетная модель является составной, то для основной операции создаются подчиненные операции.

Листинг 7.12. Создание учетной операции в базовой валюте

```
AccountNode otherEnterprise = new AccountNode("OtherEnterprise");
AccountOperation opReceiptInCash = new AccountOperation(receiptInCash,
    otherEnterprise, DateTime.Now, 1000, null);
// Перечисление подчиненных операций
foreach(AccountOperation subOperation in opReceiptInCash.SubOperations)
    Assert.AreEquals(152.54, subOperation.Sum);
```

Если в учетной модели определен тип учетного документа, то нужно создать учетный документ, связанный с учетной операцией.

Листинг 7.13. Создание учетного документа

```
AccountNode manager = new AccountNode("Manager");
AccountNode cashier = new AccountNode("Cashier");
AccountDocument docReceiptInCash = new AccountDocument(opReceiptInCash, manager
    cashier);
```

Для расчета остатков за период используется экземпляр объекта «Учетный период» **Agile.Account.AccountPeriod**, у которого вызывается метод **`Calculate`**.

Закрытие учетного периода (для блокировки изменений учетных операций за данный период) производится при вызове метода **`Close`**. Открытие учетного периода (для разрешения изменений учетных операций за данный период) производится при вызове метода **`Open`**.

Листинг 7.14. Расчет остатков за период, управление периодом

```
// Расчет остатков за учетный период
januaryMyEnt.Calculate();
// Закрытие учетного периода
januaryMyEnt.Close();
// Открытие учетного периода
januaryMyEnt.Open();
```

7.6. Интеграция с логистикой

Одним из простых способов интеграции компонента корпоративного учета с компонентом вычислительной модели логистики (см. [5]) заключается в обработке событий логистической операции на основе однозначного соответствия типов логистической операции и учетных моделей.

> Более сложным способом является разработка полноценного фасада, который объединял бы функции компонентов логистики и корпоративного учета. Такое решение необходимо в случае неоднозначного соответствия между логистическими и учетными операциями.

В логистической операции определены события завершения, отмены завершения и подтверждения отмены. Для интеграции с корпоративным учетом необходимо установить обработчики для всех этих событий.

Листинг 7.15. Подключение обработчиков событий логистической операции

```
// Обработчик события завершения логистической операции
LogisticOperation.Completed +=
    new LogisticOperation.Notifier(LogisticOperation_Completed);
// Обработчик события подтверждения отмены завершения логистической операции
LogisticOperation.Cancelling +=
    new LogisticOperation.Coordinator(LogisticOperation_Cancelling);
// Обработчик события отмены завершения логистической операции
LogisticOperation.Cancelled +=
    new LogisticOperation.Notifier(LogisticOperation_Cancelled);
```

Событие логистической операции **`Completed`** запускается при завершении ее формирования (вызове метода **`Complete`**).

> В обработчике события завершения логистической операции производится поиск учетной модели, которая соответствует заданному плану счетов и обозначению типа логистической операции.
>
> После определяется учетный узел для контрагента. Поскольку ресурсный узел в логистике соответствует учетному узлу в учете (интеграция классов на уровне данных), то создается экземпляр учетного узла по идентификатору контрагента логистической операции.
>
> Затем в общей транзакции создается учетная операция. Если в модели определен тип документа, то также создается учетный документ.

Между логистической операцией и учетной операцией устанавливается интеграционная связь с помощью класса «Связи между объектами» **Agile.Semantic.Relation** и встроенного типа связи (см. [3]).

Листинг 7.16. Обработка события завершения логистической операции

```
void LogisticOperation_Completed(LogisticOperation logisticOperation)
{
    AccountModel model = AccountModel.Find(basePlan,
        logisticOperation.Type.Sign, true);
    AccountNode contragent = new AccountNode(logisticOperation.Contragent.Id);
    using (TransactionContext tc =
        AccountComponent.Instance.DataSource.CreateTransactionContext())
    {
        AccountOperation accountOperation = new AccountOperation(model,
            contragent, DateTime.Now, operation.GetAmount(false), null);
        if (model.DocumentType != null)
        {
            AccountDocument doc = new AccountDocument(accountOperation, manager,
                cashier);
        }
        Relation.AddIntegration(logisticOperation, accountOperation);
    }
}
```

Событие логистической операции **Cancelling** запускается при попытке отмены завершения логистической операции. При обработке этого события требуется подтвердить отмену.

В обработчике события производится поиск связи интеграции между логистической и учетной операциями на основе которое создается экземпляр учетной операции. Затем возвращается подтверждение отмены, если учетный период не закрыт.

Листинг 7.17. Обработка события подтверждения отмены завершения логистической операции

```
bool LogisticOperation_Cancelling(LogisticOperation logisticOperation)
{
    Relation relation = Relation.GetIntegration(logisticOperation,
        typeof(AccountOperation), true);
    AccountOperation accountOperation =
        relation.GetTargetObject<AccountOperation>();
    return (accountOperation.Period.IsClosed == false);
}
```

Событие логистической операции **Cancelled** запускается после отмены завершения логистической операции.

В обработчике события также производится поиск связи интеграции между логистической и учетной операциями и создается экземпляр учетной операции.

В общей транзакции производится удаление учетного документа, если определен его тип, а затем удаление учетной операции и связи между логистической и учетной операции.

Листинг 7.18. Обработка события отмены завершения логистической операции

```
void LogisticOperation_Cancelled(LogisticOperation logisticOperation)
{
    Relation relation = Relation.GetIntegration(logisticOperation,
        typeof(AccountOperation), true);
    AccountOperation accountOperation =
        relation.GetTargetObject<AccountOperation>();
    using (TransactionContext tc =
        AccountComponent.Instance.DataSource.CreateTransactionContext())
    {
        if (accountOperation.DocumentType != null)
        {
            AccountDocument doc =
                new AccountDocument(accountOperation.OperationId);
            doc.Delete();
        }
        else
        {
```

```
            accountOperation.Delete();
        }
        relation.Delete();
    }
}
```

Заключение

Компонент корпоративного учета `Agile.Account.dll` является базовым решением для построения бизнес-системы организации. Определяет схемы учета в единой информационном пространстве для множества агентов в различных валютах. Это решение обеспечивает автономный учет для бизнес-приложений при распределенной системе учета, и является основной для разработки полнофункциональной системы бухгалтерского учета.

На следующем уровне системы требуются функции по инвентаризации и амортизации, составления баланса, для ведения финансового планирования и анализа. А также прикладные решения для формирования бухгалтерской и финансовой отчетности и отраслевых схем учета.

Данный компонент работает на основе инфраструктуры программной платформы ***Agile*** и является референсной бизнес-моделью для учета экономических ресурсов.

Приложение А. Схема базы данных

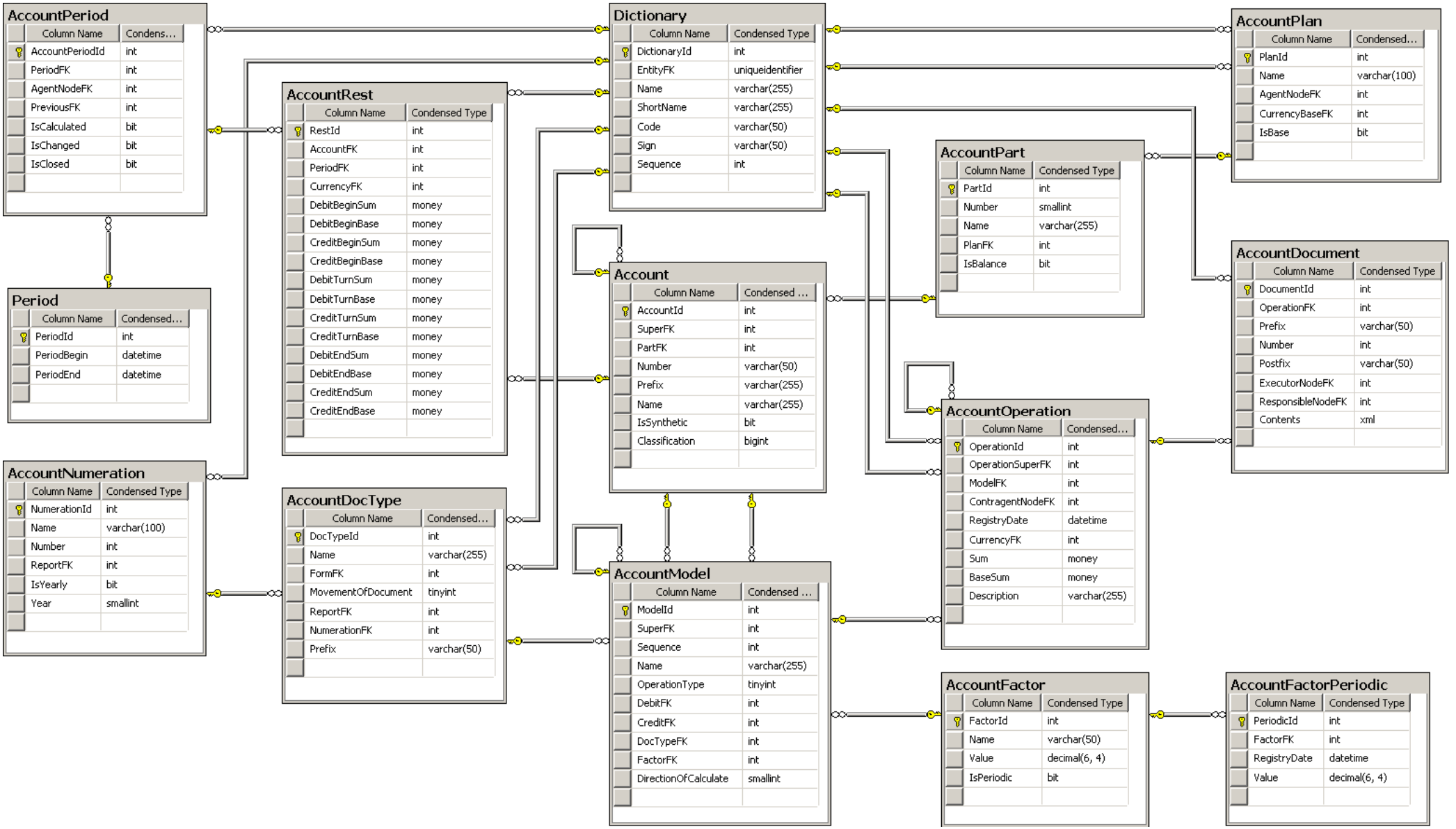
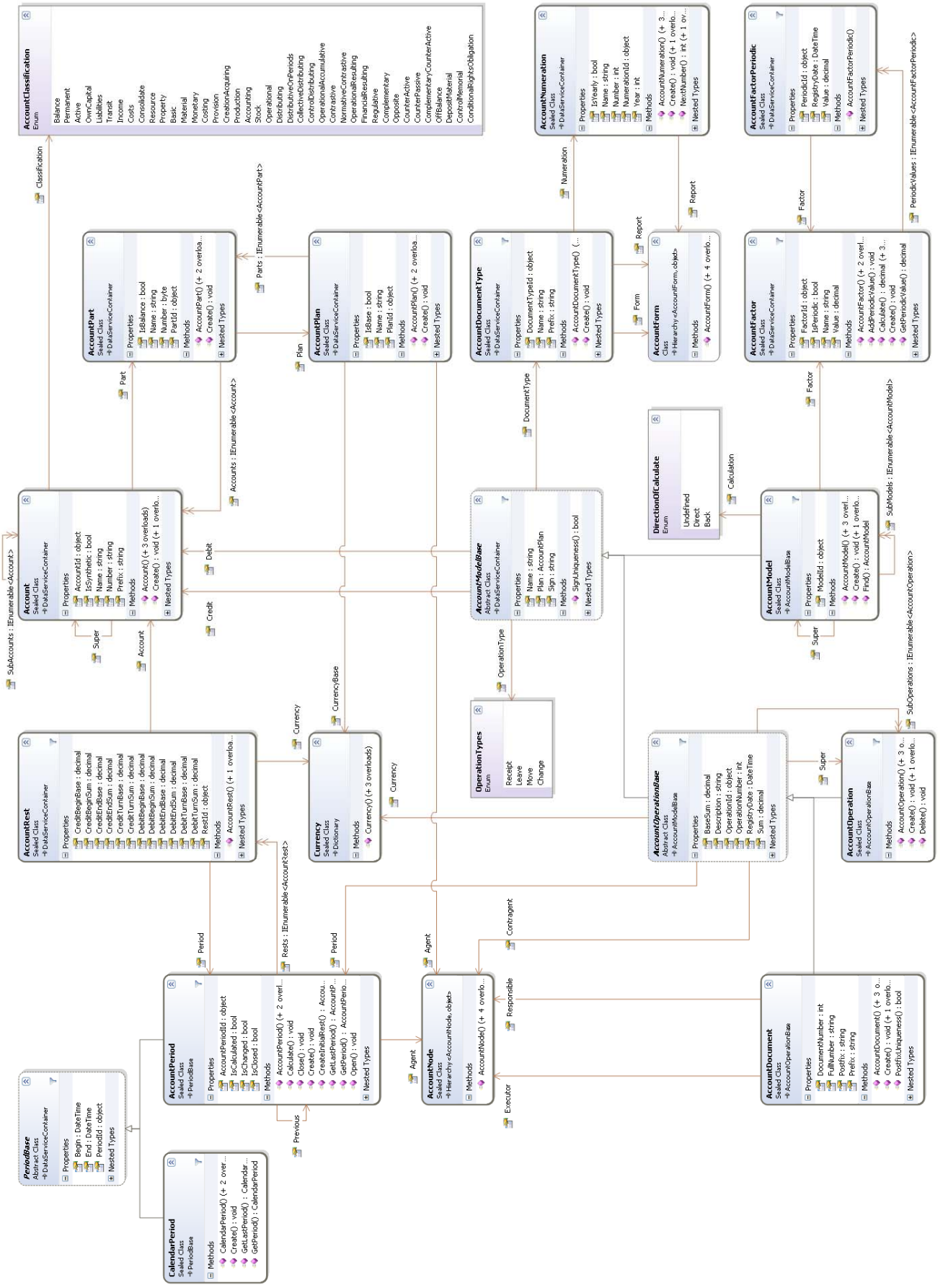


Таблица `Dictionary` принадлежит базовому типу **Agile.Semantic.Dictionary** компонента «Информационные механизмы» и используется производными от него типами компонента «Корпоративный учет».

Приложение Б. Диаграмма классов



Ссылки

1. Минюров Сергей. Сервисы данных
2. Минюров Сергей. Разработка бизнес-компонентов
3. Минюров Сергей. Информационные механизмы
4. Минюров Сергей. Инфраструктура программных компонентов
5. Минюров Сергей. Вычислительная модель логистики
6. Кутер М.И. Теория бухгалтерского учета: учебник. – 2-е изд., перераб. и доп. – М.: Финансы и статистика, 2003.