ПЛАТФОРМА ДОМИНАНТА

ПРОИЗВОДИТЕЛЬНАЯ РАЗРАБОТКА ПРИЛОЖЕНИЙ

И СРЕДА ДОВЕРЕННОГО ИСПОЛНЕНИЯ

ООО «Платформа ДОМИНАНТА», 2017 г.

ОГЛАВЛЕНИЕ

ТЕХНОЛОГИЧЕСКИЕ ХАРАКТЕРИСТИКИ

ВВЕДЕНИЕ

По мере совершенствования вычислительной техники возрастает сложность разрабатываемого программного обеспечения (ПО). Растёт объём исходного кода системных и прикладных программ. Одновременно в современных рыночных условиях к инструментам разработки предъявляется требование увеличения производительности разработки ПО.

В «Платформе ДОМИНАНТЕ», за 12 лет развития, впервые в полностью российском продукте, в комплексе реализованы самые современные подходы разработки ПО. Это объектно-ориентированное, обобщённое, событийное программирование, управляемый код. Все компоненты платформы: компилятор, виртуальная машина и фреймворк созданы «с нуля», без каких-либо заимствований стороннего программного кода.

ЯЗЫК ПРОГРАММИРОВАНИЯ

Одним из удачных языков программирования, получившим широкое распространение в мире, стал язык C. За ним последовали C++, Java, C#. Каждый из этих языков брал за основу синтаксис предшественника. Благодаря концепции управляемого кода языки Java и C# по распространённости занимают 1-е и 4-е место соответственно согласно рейтингу TIOBE. Однако в Java и C# не было включено несколько производительных парадигм предшествующего им C++.

В «Платформе ДОМИНАНТЕ» за основу принят синтаксис Java и C#. Они хорошо известны широкому кругу программистов. Дополнительно используются производительные парадигмы известные по C++, например множественное наследование.

Уникальными отличиями «Платформы ДОМИНАНТЫ» являются:

- одновременная поддержка объектно-ориентированного, обобщённого, событийного программирования, управляемого кода и множественного наследования
- языковые расширения доступа к данным
- реализация, переносящая вычислительную нагрузку на компилятор, что упрощает и ускоряет виртуальную машину
- легкая виртуальная машина, облегчающая перенос на новые архитектуры

Примеры исходников: http://platdom.ru/upload/platdom_win.zip

СОБСТВЕННАЯ АРХИТЕКТУРА

Для «Платформы ДОМИНАНТЫ» созданы собственный кроссплатформенный компилятор и виртуальная машина. Сторонние технологии не используются, центр компетенции находится в России. Это позволяет управлять всеми низкоуровневыми и аппаратными аспектами функционирования системы, оптимизировать базовую архитектуру системы для увеличения производительности, а также упрощает перенос на новые устройства.

Компоненты платформы доступны для скачивания по адресу: www.platdom.ru

ВИРТУАЛЬНАЯ МАШИНА

Архитектура системы спроектирована таким образом, чтобы перенести максимум вычислительной нагрузки на компилятор и упростить виртуальную машину. Детали приведены ниже. Исходный текст виртуальной машины занимает примерно 150 Кб. В режиме без JIT-компиляции она оперирует байт-кодом, состоящим из почти 200 команд.

МЕНЕДЖМЕНТ ПАМЯТИ – СБОРКА МУСОРА

Важной особенностью управляемого кода является автоматические выполнение одной из трудоёмких и сложных в отладке операций – высвобождение неиспользуемых участков памяти. Она называется сборкой мусора.

Способ сборки мусора, принятый в Java и C#, это вызов сборщика в момент, когда заканчивается свободная память, или когда приложение предположительно простаивает. Упрощенно рассмотрим его работу. Формируется набор корневых ссылок сканированием кадров стека и статических переменных. Используя полученные ссылки, обходится весь граф объектов. Те объекты, до которых дошёл обход, являются «действующими», они помечаются. Остальные представляют собой «мусор». На следующем проходе память, которую занимают объекты без пометки, освобождается для повторного использования. Граф должен не меняться при обходе, поэтому для сборки мусора нужны паузы в работе приложения “stop-the-world”.

Алгоритмы работы сборщика мусора усложняют виртуальную машину. Сборка мусора является ресурсоёмкой операцией. Моменты вызова сборщика непредсказуемы и сопровождаются провалами производительности. Это неприемлемо для приложений реального времени. Увеличивается расход памяти. Кроме того, деструкторы классов вызываются в момент сборки мусора, а не сразу после окончания использования объекта. Это не позволяет размещать код освобождения ресурсов, которые больше не используются, в самом логичном расположении – деструкторе класса. Это значительная проблема, что подтверждается добавлением в поздние версии языков специального синтаксиса, ‘try с ресурсами’ в Java и ‘using’ в C#. Принудительные вызовы сборщика мусора избыточны и ухудшают производительность приложения, а использование специального синтаксиса для освобождения ресурсов увеличивает трудоёмкость разработки.

Поэтому в «Платформе ДОМИНАНТЕ» реализована сборка мусора подсчётом ссылок на объект. Созданный относительно недавно компанией Apple язык Swift использует схожий принцип.

Как только количество ссылок на объект становится равным нулю, вызывается деструктор и затем освобождается занимаемая память. Операции подсчёта ссылок поддерживаются на уровне компилятора и встраиваются в байт-код.

Подсчёт ссылок – лёгкая операция. Операции подсчёта выполняются в общем потоке, они распределены по времени и не требуют пауз в работе приложения. Как только объект перестает использоваться, сразу же вызывается его деструктор и освобождается память. Это позволяет использовать при разработке такие идиомы ООП как RAII (Resource Acquisition Is Initialization) располагая код освобождения ресурсов в деструкторе.

ИСКЛЮЧЕНИЯ

В Java и C# при генерации исключения виртуальная машина просматривает таблицы исключений и на их основе принимает решение о передаче управления. Реализация этих алгоритмов усложняет виртуальную машину.

Код блока, располагающийся после исключения, игнорируется. Соответственно переменные, ссылающиеся на объекты, не очищаются. Вызовы деструкторов и удаление объектов возлагаются на сборщика мусора. Предсказать время вызова сборщика затруднительно.

«Платформа ДОМИНАНТА» гарантирует своевременность удаления неиспользуемых объектов. Поэтому в байт-коде указаны переходы для любых ситуаций времени исполнения. Такой подход переносит вычислительную нагрузку на время компиляции и алгоритмически упрощает виртуальную машину.

Компилятор прописывает переходы в байт-коде от возможной точки исключения и далее по маршруту раскрутки стека (unwind) для очистки переменных выходящих за свои области видимости. При возникновении исключения, исполнение следует по этому маршруту до появления подходящего обработчика исключения (catch).

Приведённая последовательность обработки исключений сложна для реализации в компиляторе, но обладает огромным преимуществом. Этот механизм не зависит от архитектурных особенностей среды исполнения. Будь то xNIX или Win, Intel или ARM. Обеспечивается стабильность, своевременность высвобождения неиспользуемых ресурсов и производительность.

ИСКЛЮЧЕНИЯ В КОНСТРУКТОРАХ И ДЕСТРУКТОРАХ

В отличие от C++, в «Платформе ДОМИНАНТЕ» возникновение исключений в конструкторах и деструкторах допускается и обрабатывается корректно.

ОБЪЕДИНЁННЫЕ ОБЪЕКТЫ

Для ряда задач необходимы частые создания объектов сложной структуры. Например, при обработке баз данных записи таблицы БД соответствует класс. Полями этого класса являются ссылки на классы, представляющие поля таблицы БД. При получении с сервера БД каждой записи для неё создаётся объект. Для каждого поля также создаётся по объекту, им присваиваются соответствующие значения из БД. Когда количество полей измеряется десятками или сотнями, для каждой записи соответствующее количество раз вызывается менеджер выделения памяти. Множественные выделения памяти являются ресурсоёмкой операцией.

Одним из способов повышения производительности применяемым в «Платформе ДОМИНАНТЕ» для таких случаев является использование структур для хранения данных полей. Однако в этом случае программная логика обработки разных полей помещается в класс представляющий запись.

Поэтому предпочтительным способом является использование объединённых объектов. Специальный синтаксис позволяет объявить поле класса. При создании объекта этого класса объекты, представляющие поля, будут располагаться в его адресном пространстве. Вне зависимости от сложности класса память будет выделена за один раз. Классы,

представляющие поля, используются как обычно, в них инкапсулируется логика обработки данных.

```
struct StructField
{
   public int IntPart;
   public double DoublePart;
}

class ClassField
{
   StructField Value;
   public StructField Get() { return(Value); }
}

class Record
{
   public StructField Field1; // Располагается в адресном пространстве объекта
   public ClassField Field2; // Обычная ссылка на объект в памяти
   public ClassField Field3(); // Располагается в адресном пространстве объекта
}
```

ГЕТЕРОГЕННЫЕ ВЫЧИСЛЕНИЯ

Перспективным направлением развития «Платформы ДОМИНАНТЫ» является разрабатываемая технология исполнения на вычислительных платформах с гетерогенной архитектурой (CPU+GPU). Архитектура современных GPU подсистем в настоящее время активно развивается, а их производительность быстро растёт.

Важно отметить, что поддержка предоставления вычислительной мощности GPU ресурсоёмким приложениям реализована на уровне компилятора. Изменения вносятся в кодогенератор и виртуальную машину. Это предоставляет существенные преимущества по сравнению с реализациями в прикладных библиотеках. Исчезает необходимость вносить изменения в прикладной программный код для использования гетерогенных вычислений.

ИСПОЛЬЗОВАНИЕ ПЛАТФОРМОЗАВИСИМЫХ БИБЛИОТЕК

Для обращения к API операционных систем и для вызовов функций существующих библиотек объявляются прототипы. В прототипе указывается имя файла библиотеки без расширения, имя функции библиотеки, параметры функции, а также новое имя для обращений из кроссплатформенного кода.

```
// Наше объявление
класс Файл
{
    ...
    открытые статические логическое Существует( строка ИмяФайла )
        библиотека "DominSysLibrary"
        метод "_FileExists";
    открытые статические целочисленное Размер( строка ИмяФайла )
        библиотека "DominSysLibrary"
        метод "_FileSize";
```

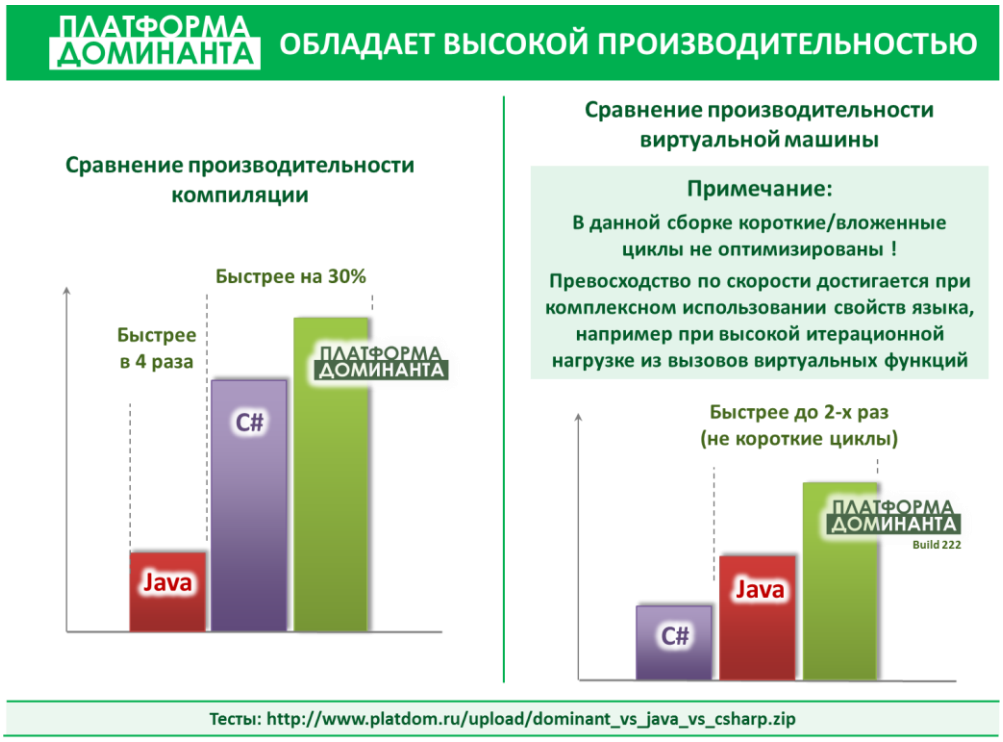
В зависимости от операционной системы, на которой исполняется прикладное приложение, подгружается соответствующая библиотека: с расширением .DLL в Windows, с расширением .SO в Linux или с расширением .DYLIB в MacOS.

Пример реализации платформозависимой прикладной библиотеки на C++:

```
extern "C" bool __export FileExists( KRTString* FileName )
{
        return( ::FileExists( FileName->String() ) );
}

extern "C" int __export FileSize( KRTString* FileName )
{

   int Handle= ::FileOpen( FileName->String(), fmOpenRead | fmShareDenyNone );
      if( Handle == -1 )
      {
          throw( KLibraryThrow( NewRTString(UnicodeString(L"Cannot open file '") +
                FileName->String() + L"'" ) ) );
      }

   int Size= ::FileSeek( Handle, 0/*offset*/, 2/*from the end of a file*/ );
      ::FileClose( Handle );
      return( Size );

}
```

ПРОИЗВОДИТЕЛЬНОСТЬ КОМПИЛЯТОРА

«Платформа ДОМИНАНТА» показывает высокую производительность при компиляции и при исполнении виртуальной машиной (**примечание**: в текущей сборке короткие/вложенные циклы не оптимизированы, а превосходство по скорости виртуальная машина показывает в сценариях комплексного использовании свойств языка, например при высокой итерационной нагрузке из вызовов виртуальных функций, как в примере).



Исходники тестов: http://www.platdom.ru/upload/dominant-vs-java-vs-csharp.zip

```
//---------------------------------------------------------------
=== Компиляция ДОМИНАНТА ===

Платформа ДОМИНАНТА, версия 3 сборка 222 beta release (BCPP x86), (c) 2002-2016,
support@platdom.ru

Timer 3.01  Copyright (c) 2002-2003 Igor Pavlov  2003-07-10
Global Time  =     0.406 = 00:00:00.406 = 100%


=== Компиляция C# ===
Microsoft (R) Visual C# Compiler version 4.6.1055.0

for C# 5
Copyright (C) Microsoft Corporation. All rights reserved.


Timer 3.01  Copyright (c) 2002-2003 Igor Pavlov  2003-07-10
Global Time  =     0.624 = 00:00:00.624 = 100%


=== Компиляция Java ===
javac 1.8.0_92

Timer 3.01  Copyright (c) 2002-2003 Igor Pavlov  2003-07-10

Global Time  =     4.383 = 00:00:04.383 = 100%


//---------------------------------------------------------------
=== Исполнение виртуальной машиной ДОМИНАНТЫ ===

Платформа ДОМИНАНТА, версия 3 сборка 222 beta release (BCPP x86), (c) 2002-2016,
support@platdom.ru

Count=88473, Res=7078400
[OK] Исполненное приложение вернуло код = 88473
Время исполнения = 0,047 сек.

Timer 3.01  Copyright (c) 2002-2003 Igor Pavlov  2003-07-10
Global Time  =     0.047 = 00:00:00.047 = 100%



=== Исполнение виртуальной машиной C# ===
Count=88473, Res=7078400

Timer 3.01  Copyright (c) 2002-2003 Igor Pavlov  2003-07-10
Global Time  =     0.671 = 00:00:00.671 = 100%



=== Исполнение виртуальной машиной Java ===
Count=88473, Res=7078400

Timer 3.01  Copyright (c) 2002-2003 Igor Pavlov  2003-07-10

Global Time  =     0.156 = 00:00:00.156 = 100%

//---------------------------------------------------------------
```

ЯЗЫК ПРОГРАММИРОВАНИЯ

ЭЛЕМЕНТЫ СИНТАКСИСА

1. Литералы
    1.1. Числовые
    1.2. Символьные
    1.3. Шестнадцатеричные
    1.4. Строковые
        1.4.1. Базовая кодировка UTF16
    1.5. Управляющие последовательности в строковых литералах
    1.6. Логические
    1.7. null
2. Переменные
    2.1. Строгая типизация переменных
    2.2. Простые типы
        2.2.1. char
        2.2.2. int
        2.2.3. double
        2.2.4. bool
    2.3. Ссылочные типы – указатели на объекты классов
    2.4. Все типы данных являются наследниками базового типа Object
    2.5. Переменные могут инициализироваться при объявлении
    2.6. Области действия локальных переменных – блок {}. Блоки могут быть вложенными.
        2.6.1. Объявляется переменная в любом месте блока. Переменная становится действительной после своего объявления
    2.7. Приведения типов
        2.7.1. Явные
        2.7.2. Неявные автоматические
3. Операции
    3.1. Арифметические '+ - * / %'
    3.2. Инкремент, декремент '++ --'
    3.3. Отношения '== != > < <= >='
    3.4. Логические '&& ||'
        3.4.1. Вторая часть оператора && || не выполняется если результат определен первой частью выражения
    3.5. Поразрядные '& |'
    3.6. Присваивания '='
        3.6.1. Цепочки операторов присваивания
    3.7. Составные операторы присваивания '+= -= *= /= %='
    3.8. Тернарный условный оператор '? '
4. Управляющие операторы
    4.1. if, else, else if
    4.2. switch
    4.3. for
        4.3.1. break
        4.3.2. continue
    4.4. return

ПРЕИМУЩЕСТВА ЯЗЫКА ПРОГРАММИРОВАНИЯ

МНОЖЕСТВЕННОЕ НАСЛЕДОВАНИЕ

Одной из наиболее прогрессивных парадигм разработки используемых в C++ является множественное наследование. Оно позволяет классу стать наследником нескольких родительских классов. Наследуются все члены класса: данные, статические и виртуальные функции.

Эта парадигма не вошла в Java и C# в полном виде. В этих языках полноценное наследование возможно только от одного класса. У остальных предков должны отсутствовать данные, а их функции должны быть только объявлены, без реализаций. Такие предки называются интерфейсами.

Подобный подход приводит к возникновению однотипных по функционалу участков кода при каждом использовании интерфейса. Появляются copypast'ы при реализации функций интерфейса. Дублирование кода ухудшает структуру программы, является источником ошибок при поддержке кода и снижает производительность разработки.

При отсутствии множественного наследования и его поддержки в runtime'е, из-за упрощения RTTI работа по созданию кодогенератора языка сокращается на треть. А разработчики в результате лишились мощного и производительного инструмента.

Поэтому в «Платформе ДОМИНАНТА» реализовано полноценное множественное наследование аналогичное C++.

Таблицы виртуальных функций и коррекции указателя 'this' при приведении объекта к базовым либо производным классам поддерживаются при помощи RTTI.

```
class A
{
   public int IntA;
   public A( int __IntA ) { IntA= __IntA; }
   public int FuncA() { return( IntA ); }
   public virtual int FuncVirt() { return( IntA ); }
}

class B
{
   public int IntB;
```

```
   // B() {} // Constructor by default
   public int FuncB() { return( IntB ); }
   public virtual int FuncVirt() { return( IntB ); }
}

class C: A, B
{
   public int IntC;
   public C(): A(1), B() { IntB= 2; IntC= 3; }
   public int FuncC() { return( A.FuncA() + B.FuncVirt() + C.FuncVirt() + IntC ); }
   public override int FuncVirt() { return( IntC ); }
}
```

ОБОБЩЕННОЕ ПРОГРАММИРОВАНИЕ

В Java и C# механизмы обобщенного программирования были упрощены по сравнению с C++. Для соответствующих конструкций этих языков – дженериков создаются общие реализации в байт-коде. Это требует указания ограничений на допустимые типы параметров дженериков. Нельзя использовать статические функции и перегруженные операции, нельзя наследовать от параметра дженерика. Выбор перегруженной функции происходит для типа, указанного в качестве условия дженерика, вне зависимости от реально подставленного. Существует множество других ограничений.

В «Платформе ДОМИНАНТЕ» для обобщенного программирования используются шаблоны – подход, принятый в C++. Тело шаблона компилируется для каждого набора параметров. Отсутствуют ограничения на тип параметра шаблона. При использовании обобщённого программирования можно использовать все свойства языка. Нет ограничений на доступ к характеристикам подставленного типа.

```
//---[1]-----------------------------
class A1
{
   A1( int Val ) {}
   public int fn() { return( 1 ); }
}

class C<T> { public int Test() { T t= T(1); return( t.fn() ); } }

class Program
{
   static int Startup( string CommandLine )
   {
      C<A1> ca1= C<A1>();
      return( ca1.Test() );
   }
}

//---[2]-----------------------------
class A2<T>
{
    public int fn( T t1, T t2 )
    {
        return (t1 + t2);
    }
}

//---[3]-----------------------------
class List<ItemType>
```

```
{
    ItemType* Storage;
}
```

ПРАВИЛЬНАЯ АРХИТЕКТУРА

В распространённых на сегодняшний день языках программирования с управляемым кодом Java и C# существуют недостатки архитектуры.

Например, можно вызвать виртуальный метод производного класса из конструктора родительского, хотя производный класс в этот момент ещё не инициализирован. Это приводит к аварийной остановке программы при исполнении.

Java:

```
package javaapplication1;

class A {
        public A() { VirtFn(); }
        public void VirtFn() { System.out.println("It's OK"); } }

class B extends A {
    public  void VirtFn() { throw new RuntimeException("CRASH!"); } }

public class JavaApplication1 {
    public static void main(String[] args) { A a = new B(); } }


Output>>>
Exception in thread "main" java.lang.RuntimeException: CRASH!
```

C#:

```
namespace ConsoleApplication1
{
    class A {
        public A() { VirtFn(); }
        public virtual void VirtFn() { Console.WriteLine("It's OK"); } }

    class B : A {
        public override void VirtFn() { throw new Exception("CRASH!"); } }

    public class Test { public static void Main() { A a = new B(); } }
}


Output>>>
Необработанное исключение типа "System.Exception" в ConsoleApplication1.exe
Дополнительные сведения: CRASH!
```

В «Платформе ДОМИНАНТЕ» этот и другие архитектурные недостатки языков-предшественников исправлены:

```
namespace App1
{
class A {
   public A() { VirtFn(); }
   public virtual VirtFn() { Console.PrintLine("The Platform DOMINANT is running"); }
}

class B: A {
   public override VirtFn() { throw( Exception("Java && C# are crashed!") ); } }

class C {
   static int Startup( string CommandLine ) { A a = B(); } }
}


Output>>>
The Platform DOMINANT is running
```

ЛОКАЛИЗАЦИИ ЯЗЫКА ПРОГРАММИРОВАНИЯ

Основная локализация «Платформы ДОМИНАНТА» – английская, традиционная для языков программирования.

Подход, использованный российским разработчиком 1С – русский синтаксис языка программирования, доказал свою жизнеспособность. В промышленных проектах разработки ПО всегда участвуют специалисты в предметной области, не являющиеся профессиональными программистами. Они выступают в роли постановщиков задач. Тем не менее, начальное представление о программировании ими, как правило, уже получено в ВУЗе. Использование русского синтаксиса позволяет снизить сложность понимания программной логики такими специалистами и упростить их обучение. Специалисты привлекаются для разработки трудоёмких частей проекта требующих знания предметной области, например для создания запросов данных и формирования отчётов.

Поэтому в «Платформе ДОМИНАНТЕ» доступна русская локализация. Существует функция автоматического перевода с языка на язык.

```
//
// Код лёгок для ЧТЕНИЯ и ПОНИМАНИЯ
//

если( ТипЭлектрода.назначено && ТипЭлектрода.Аустенитный() &&
ТП.НаименееСвариваемаяСталь().СтальЗакаливающаяся() )
{
   ПоказатьСообщение(
      "Применение аустенитных электродов при сварке закаливающихся сталей "
      "допускается только для кольцевых и угловых стыков." );
}

если( ТП.Сталь1.Марка=="10Х17Н13М2Т" || ТП.Сталь1.Марка=="08Х17Н15М3Т" )
{
   если( ТП.ТребПоМежкристалКорроз )
   {
      если( ТП.ТемпПриЭксплуатации>=-196 && ТП.ТемпПриЭксплуатации<=+700 )
```

```
      {
         если( ТП.ТемпПриЭксплуатации <= +450 )
         {
            ЗапросСварМат= ЭлектродДляВысоколегирСталей10052[
                  условие Тип=="Э-02Х20Н14Г2М2" ] |
               МаркаЭлектродаДляРДС[ условие Марка=="ОЗЛ-20" ] |
               ЭлектродДляВысоколегирСталей10052[ условие Тип=="Э-02Х19Н18Г5АМ3" ] |
               МаркаЭлектродаДляРДС[ условие Марка=="АНВ-17" ] |
               ЭлектродДляВысоколегирСталей10052[ условие Тип=="Э-08Х17Н8М2" ] |
               МаркаЭлектродаДляРДС[ условие Марка=="НИАТ-1" ];
         }
         иначе
         {
            ЗапросСварМат= ЭлектродДляВысоколегирСталей10052[
                  условие Тип=="Э-09Х19Н10Г2М2Б" ] |
               МаркаЭлектродаДляРДС[ условие Марка=="НЖ-13" ];
         }
      }
   }
}

РассчитатьНормы();
```

```
//---------------------------------
// English:
//
class A
{
   //
   private int FuncPriv() { return( 1 ); }
   protected int FuncProt() { return( 2 ); }
   public int FuncPubl() { return( 3 ); }
   //
   private static int FuncStatPriv() { return( 4 ); }
   protected static int FuncStatProt() { return( 5 ); }
   public static int FuncStatPubl() { return( 6 ); }
   //
   protected virtual int FuncVirtProt() { return( 8 ); }
   public virtual int FuncVirtPubl() { return( 9 ); }
}

// Russian:
//
// И такие языковые конструкции непрофессиональным программистом ЧИТАЮТСЯ легче
//
//---------------------------------
класс A
{
   //
   закрытые целочисленное ФункЗакр() { возврат( 1 ); }
   защищенные целочисленное ФункЗащ() { возврат( 2 ); }
   открытые целочисленное ФункОткр() { возврат( 3 ); }
   //
   закрытые статические целочисленное ФункСтатЗакр() { возврат( 4 ); }
   защищенные статические целочисленное ФункСтатЗащ() { возврат( 5 ); }
   открытые статические целочисленное ФункСтатОткр() { возврат( 6 ); }
   //
   защищенные виртуальные целочисленное ФункВиртЗащ() { возврат( 8 ); }
   открытые виртуальные целочисленное ФункВиртОткр() { возврат( 9 ); }
}
```

ЯЗЫКОВЫЕ РАСШИРЕНИЯ ЗАПРОСОВ ДАННЫХ

Примером технологии, эффективно реализовать которую может разработчик компилятора, является Объектно-реляционное отображение (ORM – Object-Relational Mapping).

ORM связывает базы данных с концепциями объектно-ориентированного программирования, создавая «виртуальную объектную базу данных». Технология устраняет «семантический разрыв» между языком взаимодействия с сервером БД, таким как SQL, и объектно-ориентированным языком программирования. Программист взаимодействует с объектами как обычно, а ORM берёт на себя обмен данными с сервером БД, автоматически генерируя SQL.

Однако как в случае непосредственного использования SQL, так и при посредничестве ORM, без поддержки на уровне компилятора нельзя автоматически установить ошибку несоответствия структуры запроса структуре базы данных. Запросы на сервер БД формируются в виде текстовой строки, и ошибка проявляется уже после компиляции во время работы приложения, когда запрос будет передан на сервер и отвергнут. А поскольку многие запросы выполняются в зависимости от условий, при неполном покрытии тестированием, ошибки зачастую проявляются уже после начала эксплуатации приложения.

В «Платформе ДОМИНАНТЕ» языковые расширения запросов данных поддерживаются компилятором. Ошибки запросов выявляются сразу же, на этапе компиляции. Использование языковых расширений позволяет ускорить программирование логики обработки данных и уменьшить в несколько раз себестоимость её создания.

ИСПОЛЬЗОВАНИЕ ЯЗЫКОВЫХ РАСШИРЕНИЙ

1. Проектирование схемы данных

В визуальном конструкторе создаётся схема данных. Конструируются простые и составные таблицы, указываются поля и связи. Для элементов схемы указываются их логические имена, принятые в предметной области. Логические имена используются в дальнейшем для создания запросов данных в терминах предметной области.

По схеме данных автоматически генерируются объекты в базе данных.

2. Программирование прикладной логики приложения и компиляция

По схеме данных автоматически создаются классы, соответствующие записям таблиц БД. Разработчик программирует логику обработки данных реализацией методов этих классов.

При компиляции языковых расширений для запросов данных, автоматически проверяется соответствие структуры запросов структуре базы данных.

Управление транзакциями осуществляется посредством вызовов методов фреймворка ORM.

3. Эксплуатация приложения

В соответствии с откомпилированными запросами данных автоматически генерируются SQL операторы целевой СУБД. В результате их исполнения сервером СУБД, на клиента

доставляются записи таблиц. Для каждой записи БД создаётся объект соответствующего класса.

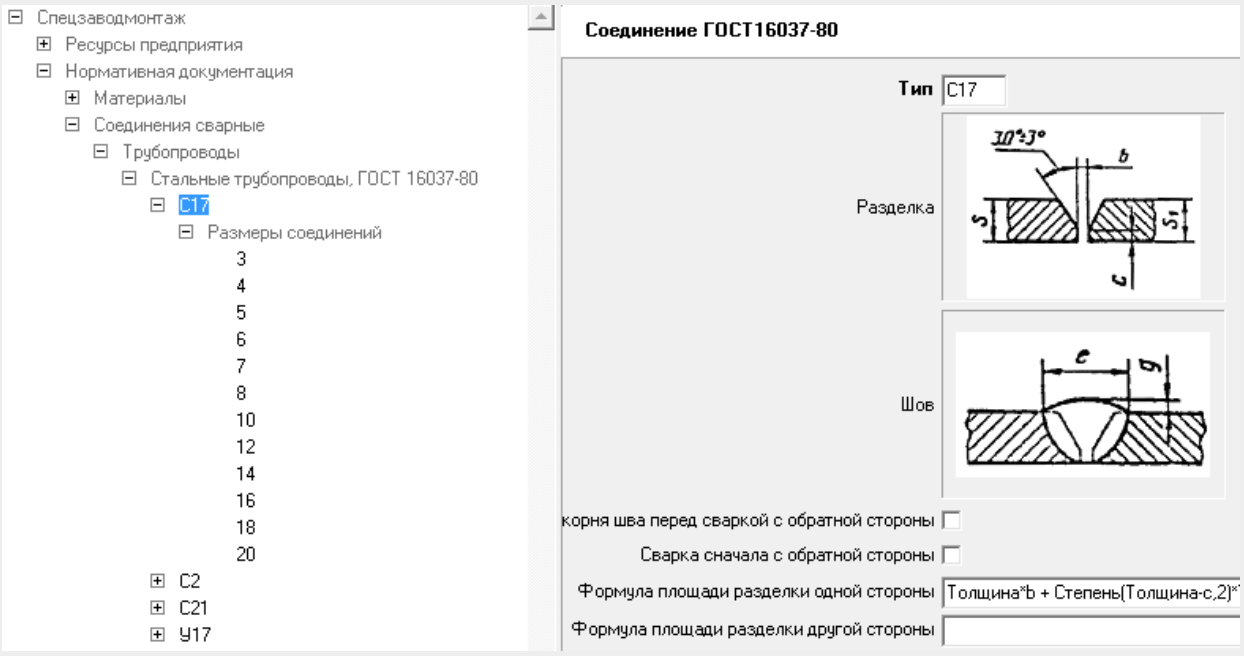
При изменении значений полей объекта автоматически генерируются SQL операторы изменения данных на сервере СУБД.

ПРЕИМУЩЕСТВА ORM

Запросы данных в терминах предметной области

Написание запросов данных в терминах предметной области интуитивно и производительно.

```
перем ЗапросТипоразмеры=

   СоединениеГОСТ16037_80[
      условие Тип != {Техпроцесс.Соединение.Тип}
      курсор КурсорСоединение ]
   ->
   РазмерыСоединения[
      условие {Техпроцесс.ТолщинаСтенкиТрубы} >= s_min
      сортировка s_min, s_max
      курсор КурсорТипоразмеры ];


//--- Результат ---
```



Результатом выполнения запроса является дерево. Объекты, представляющие записи, связаны как листья дерева. Фреймворк предоставляет методы обхода дерева. Данные кешируются на стороне клиента.

Объединение, пересечения и разность деревьев

В результате выполнения запроса данных формируется дерево. Важным преимуществом ORM «Платформы ДОМИНАНТЫ» является возможность выполнения объединения, пересечения и разности деревьев. Знаки операций |, &, ^ соответственно.

```
ЗапросСварМат=
   ЭлектродДляВысоколегирСталей10052[ условие Тип=="Э-08Х20Н9Г2Б" ] |
   МаркаЭлектродаДляРДС[ условие Марка=="ЦЛ-11" || Марка=="ОЗЛ-7" || Марка=="Л-40М"
      || Марка=="АНВ-23" ] |
   ЭлектродДляКонструкцИТеплоустСталей9467[ условие Тип=="Э-42" ];
```

Операции выполняются на стороне сервера БД, на клиента передаётся только результат.

Компиляция с учётом структуры БД

Уникальным преимуществом ORM «Платформы ДОМИНАНТЫ» является учёт компилятором структуры базы данных при обработке языковых расширений. При несоответствии запроса структуре БД компиляции будет прервана с ошибкой.

Сложные связи с поддержкой целостности на уровне БД

В процессе промышленного проектирования схем данных часто возникает потребность в указании связи поля с записью одной из набора таблиц. Однако реляционные СУБД допускают создание для одного поля только одного внешнего ключа. Для таких связей приходится отказываться от средств поддержания ссылочной целостности средствами реляционной СУБД, перенося поддержание на уровень прикладного приложения. В этом случае логическая целостность данных БД становится уязвима перед ошибками приложения или злонамеренными модификациями. Либо приходится усложнять структуру таблиц, создавая по дополнительному полю на каждый внешний ключ. В этом случае замедляется обработка данных, а в случае заполнения более чем одного из этих полей нарушается логическая целостность данных.

Поэтому отличительной особенностью «Платформы ДОМИНАНТЫ» является поддержание ссылочной целостности сложных связей за счёт переноса логики контроля в автоматически генерируемые триггеры и представления. Это позволяет надёжно поддерживать ссылочную целостность средствами СУБД, обеспечить безопасность путём разграничения прав доступа на уровне СУБД и увеличить производительность.

Составные таблицы

Ещё одним мощным и востребованным инструментом проектирования схем данных являются составные таблицы, каждая запись которых состоит из части, находящейся в общей таблице, и части располагающейся в другой таблице. Конкретное сочетание выбирается в момент создания записи. Составные таблицы состоят из нескольких физических, поэтому для упрощения работы автоматически создаются представления (views). Проверка на допустимость создания записи и ссылочная целостность поддерживаются на стороне сервера БД через автоматически создаваемые триггеры.

ОБЩАЯ ЛОГИКА РАБОТЫ ORM

ПРИМЕР ИСПОЛЬЗОВАНИЯ ЯЗЫКОВЫХ РАСШИРЕНИЯ ЗАПРОСОВ ДАННЫХ

```
//
// Код лёгок для ЧТЕНИЯ и ПОНИМАНИЯ
//

//-----------------------------------------------------
ЗапросТипоразмеры=
   СоединениеГОСТ16037_80[
      условие Тип != {ТП.Соединение.Тип}
      курсор КурсорСоединение ]
   ->
   РазмерыСоединения[
      условие {ТП.ТолщинаСтенкиТрубы} >= s_min
      сортировка s_min, s_max
      курсор КурсорТипоразмеры ];


//-----------------------------------------------------
ЗапросСварМат=
   ЭлектродДляКонструкцИТеплоустСталей9467[ условие Тип=="Э-42А" || Тип=="Э-46А"]
   |
   МаркаЭлектродаДляРДС[ условие Марка=="УП-1/55" ];


//-----------------------------------------------------
если( ТП.Сталь1.Марка=="10Х17Н13М2Т" || ТП.Сталь1.Марка=="08Х17Н15М3Т" )
```

```
{
    если( ТП.ТребПоМежкристалКорроз )
    {
        если( ТП.ТемпПриЭксплуатации>=-196 && ТП.ТемпПриЭксплуатации<=+700 )
        {
            если( ТП.ТемпПриЭксплуатации <= +450 )
            {
                ЗапросСварМат=
                   ЭлектродДляВысоколегирСталей10052[ условие Тип=="Э-02Х20Н14Г2М2"||
                      Тип=="Э-02Х19Н18Г5АМ3" || Тип=="Э-08Х17Н8М2" ] |
                   МаркаЭлектродаДляРДС[ условие Марка=="ОЗЛ-20" || Марка=="АНВ-17"||
                      Марка=="НИАТ-1" || Марка=="ЦЛ-4" ];
            }
            иначе
            {
                ЗапросСварМат=
                   ЭлектродДляВысоколегирСталей10052[условие Тип=="Э-09Х19Н10Г2М2Б"]|
                   МаркаЭлектродаДляРДС[ условие Марка=="НЖ-13" ];
            }
        }
    }


//-------------------------------------------------
цикл(
   ЗапросОперации= Техпроцесс[условие Контекст=={ТП}]->ОперацииВсе[курсор Операция];
   ЗапросОперации.курсоры.Операция.назначено;
   ЗапросОперации.курсоры.Операция= ЗапросОперации.курсоры.Операция.СледующийУзел() )
{
   если( ЗапросОперации.курсоры.Операция.ТипУзла() == СваркаРД )
   {
      если( ЗапросОперации.курсоры.Операция.Электрод.назначено )
      {
         прервать;
      }
   }
}
```