ТЕХНОЛОГИЯ ПРОИЗВОДСТВА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

УДК: 004.4'2

ПРИМЕНЕНИЕ МЕТА-ИНФОРМАЦИИ В C++ И КОМПОНЕНТ-ОРИЕНТИРОВАННОГО ПОДХОДА ДЛЯ РАЗРАБОТКИ ИГРОВЫХ ПРИЛОЖЕНИЙ

Статья поступила в редакцию 07.10.2013, в окончательном варианте 12.10.2013.

Алимов Александр Александрович, аспирант, Волгоградский государственный технический университет, 420010, Российская Федерация, г. Волгоград, пр. им. Ленина, 28, тел.: +7(906)173-24-80, e-mail: velorth.avelies@gmail.com

Шабалина Ольга Аркадьевна, доцент, Волгоградский государственный технический университет, 420010, Российская Федерация, г. Волгоград, пр. им. Ленина, 28, тел.: +7(919)795-89-43, e-mail: o.a.shabalina@gmail.com

В статье рассматриваются вопросы эффективности применения компонент-ориентированного подхода при разработке игровых приложений. Основное внимание уделено анализу возможностей конструирования игровых объектов на основе декларативных описаний в процессе выполнения. Проведен анализ существующих программных архитектур игровых приложений, распределенных и мультиагентных систем. Рассмотрены возможные способы представления и описания компонентов, а также механизмов их взаимодействия. С учетом результатов анализа разработаны компонент-ориентированная архитектура приложения и обобщенная модель игровых объектов. Предложен способ описания декларативного шаблона модели-компонента для задания параметров модели и протокола ее взаимодействия с другими объектами приложения. Авторами реализован C++ фреймворк для работы с компонентами, основанный на использовании мета-информации о классах и функциях. Этот фреймворк был применен для разработки демонстрационного игрового приложения (ДИП). Выполнено сравнение вычислительной эффективности ДИП с приложением, построенным по классической архитектуре, где каждый новый класс объектов является наследником базового, расширяет и/или переопределяет его методы. Сделан вывод, что предлагаемая архитектура для игровых программ может быть предпочтительной.

Ключевые слова: разработка игр, программирование, C++, шаблонное мета-программирование, компонент-ориентированный подход, декларативное программирование, программные библиотеки

APPLICATION OF META-DATA IN C++ AND COMPONENT-ORIENTED APPROACH IN GAME DEVELOPMENT

Alimov Alexander A., post-graduate student, Volgograd State Technical University, 28 Lenin avenue, Volgograd, 400005, Russian Federation, tel: +7 906 173 24 80, e-mail: velorth. avelies@gmail.com

Shabalina Olga A., Associate Professor, Volgograd State Technical University, 28 Lenin avenue, Volgograd, 400005, Russian Federation, tel: +7 (919) 795-89-43, e-mail: o.a.shabalina@gmail.com

The article examines the effectiveness of component-oriented approach in game development. The focus is on the analysis of the possibilities of constructing objects game based on declarative descriptions in the process of implementation. There is an analysis of existing software architectures gaming applications, distributed and multi-agent systems. The possible ways to present and describe the components and the

mechanisms of their interaction are examined. Developed component-oriented application architecture and generalized model of game objects are under review. A method of declarative template definitions of components for setting up model parameters and interaction protocol with other application objects is proposed. Authors implemented C++ framework for managing components based on class meta-data. This framework was used for development of demonstrative game application (DGA). DGA was compared with application based on classic architecture where each new object class is derived from base and extends or overrides its methods. There is a conclusion that proposing architecture is more prefered for game applications.

Keywords: programming, C++, template meta-programming, component-oriented approach, game development, declarative programming, application frameworks

Введение. Проблематика разработки программных средств, реализующих различные игры, сейчас является достаточно актуальной – в силу высокой востребованности таких разработок. Любой объект в виртуальном мире игры может быть представлен бесконечным количеством различных моделей, каждая из которых реализует определенный аспект поведения объекта. В игровом приложении необходимо одновременное использование нескольких представлений для каждого объекта. Модели объекта должны быть синхронизированы между собой, например координаты узлов графической сцены должны соответствовать координатам физических тел, а анимации – выполняемым действиям. Классический подход к проектированию игровых программ предполагает, что синхронизацией моделей занимается определенный алгоритм. В объектно-ориентированном программировании этот алгоритм может изменяться в зависимости от типа игрового объектаКомпонент-ориентированный подход допускает возможность изменения состава моделей, представляющих объект, во время выполнения. Каждая модель представляется отдельным компонентом. При таком подходе невозможно заранее определить алгоритм синхронизации данных между моделями. Существуют различные подходы для реализации обмена данными, сообщениями и результатами вычислений между компонентами.

Целью данной работы была разработка фреймворка, позволяющего описывать типы компонентов игровых объектов на языке **C++** и создавать (инстанцировать) экземпляры компонентов по описаниям, выполненным на языке **XML**. Для этого необходимо было разработать модель представления компонентов и игровых объектов, определить механизм взаимодействия компонентов и реализовать алгоритмы сериализации и десериализации компонентов.

1. Анализ существующих работ по теме статьи. Разделение программы на независимые модули – это стандартная и хорошо зарекомендовавшая себя практика. Как правило, в модули объединяют функции с общим назначением, такие как обработка графики, «физики объектов» или организации пользовательского ввода [7]. Для реализации модели игрового объекта необходимо использовать все модули приложения, сохраняя при этом минимальное количество зависимостей между ними. В ряде случае эту проблему бывает трудно решить. В определенный (критический) момент игровые объекты становятся слишком сложными и реализующий их программный код теряет гибкость и масштабируемость. Эта проблема хорошо проявляется в проектах, демонстрирующих алгоритмы искусственного интеллекта (ИИ), таких как ORTS [5] или OpenSteer [12]. Графика и ИИ зависят друг от друга таким образом, что невозможно разделить их и использовать независимо в других проектах. Компонент-ориентированный подход позволяет решить эту проблему за счет инкапсуляции аспектов поведения в отдельных компонентах и предоставления механизмов межкомпонентого взаимодействия.

Такой подход обычно применяется одновременно с «разработкой на основе данных» (Data-driven developement). Все модели игрового объекта должны быть декларативно описаны в виде некоторой структуры данных. При этом необходимо, чтобы описания моделей

могли быть загружены в приложение или сгенерированы во время его выполнения. Сгенерированные модели также должны быть доступны сторонним программам, например редакторам уровней.

В [9] описывается архитектура системы основанной на компонентах и представление различных аспектов поведения системы, в которой содержатся описания шаблонов моделей, позволяющих создавать и настраивать различные типы моделей. Палитра шаблонов позволяет задать общие свойства для множества моделей определенного класса.

Во фреймворке MALEVA [4] реализовано явное разделение потоков управления и потоков данных, соединяющих компоненты системы. Авторы фреймворка реализовали два вида таких соединений: порты данных и управляющие порты. Порты данных позволяют компонентам разделять общие данные, а порты управления используются для подачи управляющих сигналов и сообщений. В статье также содержатся примеры элементарных и иерархических моделей, построенных с использованием фреймворка. Предложенный подход делает модели более прозрачными для понимания за счет явного указания зависимостей вместо использования скрытого состояния.

В работе [10] описывается подход, основанный на использовании технологии рефлексии. В предлагаемом программном шаблоне используется два уровня мета-информации для разделения бизнес-логики, временных ограничений и правил регуляции. Технология рефлексии позволяет использовать мета-информацию об объектах программы, предоставляемую средой выполнения. Использование этого механизма дает возможность организовать динамическую компоновку компонентов объекта.

В статье [8] описаны механизмы взаимодействия между объектами. Наиболее эффективными с точки зрения производительности являются прямые вызовы методов объектов. В языках близких к аппаратной платформе, таких как **C/C++** и **D**, вызовы методов могут быть разрешены на этапе компиляции и преобразованы в машинные инструкции. Полиморфные вызовы в высокоуровневых языках приводят к дополнительным расходам на поиск метода в иерархии наследования. Некоторые языки (**Java, C#**) предоставляют также механизм рефлексии, позволяющий обращаться к полям классов с использованием строковых идентификаторов, а иногда и конструировать новые классы и методы в процессе выполнения. Очевидно, что вызов методов с использованием рефлексии требует еще б***о***льших накладных расходов, так как часть операций с этапа компиляции переносится на этап выполнения. Рефлексия создает уязвимость для операций рефакторинга, так как автоматический рефакторинг не может отследить строковые ссылки на объекты программы. Еще одним методом организации взаимодействия является шаблон проектирования «цепочка ответственности». В качестве примера реализации шаблона можно привести сигналы и слоты **Qt**, а также события в языке **C#**. Авторы статьи предлагают использовать этот шаблон одновременно с дескрипторами свойств объектов для организации связывания компонентов объекта во время выполнения.

Компонент-ориентированный подход является более гибким, чем классическое объектно-ориентированное программирование, поскольку позволяет более удобно изменять модели поведения объектов в процессе выполнения. Например, способ проверки выполнения заданий, описанный в работе [3], может осуществляться с помощью моделирования выполнения тестируемой программы. К каждому интерактивному объекту может быть добавлен компонент, связанный с тестируемым кодом и выполняющий соответствующие инструкции.

Из соображений производительности большинство игровых приложений и библиотек написаны на языках **C** и **C++**, а для декларативных описаний моделей часто используются встраиваемые языки (**Lua**) и языки описания данных (**XML**, **Json**).

Средства моделирования обычно предоставляют проблемно-ориентированные декларативные языки для написания определений моделей. Эти языки, как правило, не такие

производительные, как классические императивные, поэтому некоторые инструменты часто предоставляют гибридное лингвистическое обеспечение. Хорошим примером гибридного языка является **XAML** [11], который представляет собой подмножество языка **XML**. Каждый элемент в **XAML** описывает ресурс, элемент интерфейса или свойство. Свойства объекта, имеющие примитивный тип, могут быть описаны атрибутом элемента, свойства сложных типов – вложенным элементом. **XAML** также предоставляет механизмы для задания ссылок на ресурсы и источники данных. Эти механизмы применяются для связывания декларативной разметки и императивного кода, описывающего поведение элемента интерфейса.

Популярный набор инструментов для разработки игр Unity 3D также разработан с использованием компонент-ориентированного подхода [13]. Компоненты Unity 3D взаимодействуют через систему сообщений. Каждая посылка сообщения – это вызов метода объекта по его имени во всех компонентах объекта. Этот механизм схож с широковещательной рассылкой, так как адресат может быть не один. Все пользовательские расширения пишутся на языках **C#**, **Java Script** или **Boo** и расширяют базовый класс *Component*. Все компоненты сериализируются в бинарные или текстовые файлы.

Итак, компонент-ориентированный подход позволяет заменить наследование в объектно-ориентированных программах на композицию и, таким образом, уменьшить связанность кода. Для этого необходимо разработать модели представления объектов игрового приложения и компонентов, из которых состоит объект. Существуют различные методы организации компонентов, но все они требуют хорошей поддержки мета-информации. Реализация концепции на языке **C++** (для использования в игровых приложениях) требует разработки библиотеки, реализующей предоставление мета-информации компонентов.

2. Концепция фреймворка

Модель объекта. Приложение состоит из объектов игрового мира и системных объектов (невидимых для игрока). К объектам игрового мира относятся ландшафты, персонажи, интерактивные и статические объекты, задания и т.д. К системным объектам относится экземпляр класса приложения, система управления игровыми объектами (сцена), система ввода. Каждый игровой объект может быть представлен кортежем

$$GO = (uid, name, M),$$

где ***uid*** – уникальный численный идентификатор объекта; ***name*** – строковое имя для использования в декларативной разметке; ***M*** – множество компонентов-моделей. Компоненты могут взаимодействовать между собой двумя способами. Первый – разделение данных, второй – обмен сообщениями. Каждый компонент может находиться в двух состояниях: запрещенном и разрешенном. Работа компонента осуществляется только в разрешенном состоянии.

Разделение данных между компонентами не так тривиально, как может показаться. Рассмотрим это на примере координат объекта, которые разделяются между узлом графической сцены и телом физической модели. После каждого шага физического моделирования координаты графического узла тоже должны измениться. Физическое тело выступает источником данных для графического узла, но при этом размерность графических координат может не совпадать с физическими, а некоторые объекты и вовсе не имеют физической модели. Поэтому в качестве решения нами использован двусторонний объект-делегат (шаблон цепочка ответственности), связывающий компоненты объекта, имеющие общий источник данных.

Обмен сообщениями между компонентами осуществляется с помощью механизма сигналов и слотов. При переходе компонента в запрещенное состояние все сигналы отключаются, а при переходе в разрешенное – подключаются обратно. Соединения сигналов и слотов описываются декларативно в XML-файле. Также может осуществляться широковещательная рассылка сообщений. Метод компонента вызывается по его имени с соответствующими параметрами.

Для описания игровых объектов разработана графическая нотация. Структура сборки компонентов отображена на рис. 1. Прямоугольники со скругленными углами показывают компоненты, где с левой стороны перечислены слоты, а с правой – сигналы. Линии между сигналами и слотами обозначают соединения. Сигнатуры соединений подписаны под линиями. Сигналы объекта-сборки показаны на левой стороне диаграммы. Обмен данными на диаграмме не показан.

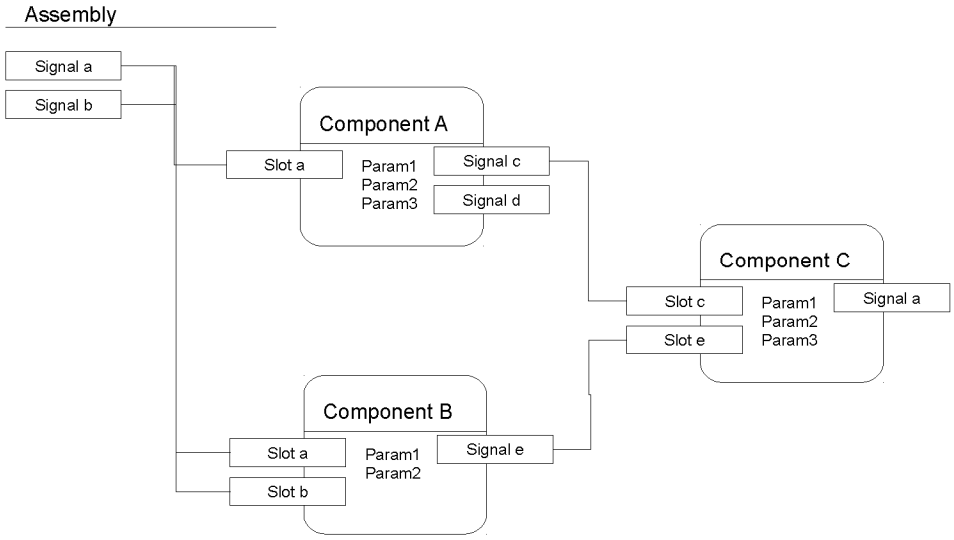

Рис. 1. Схема компонентов и соединений

Лингвистическое обеспечение. Для описания компонента объекта как модели удобно использовать две парадигмы программирования: императивную и декларативную. Код на императивном языке описывает алгоритм работы класса компонента, а код на декларативном языке – экземпляр компонента. В качестве императивного языка используется **C++**. Классы компонентов реализуются как наследники от базового класса с набором ограничений. Компонент может наследоваться только от одного класса. Класс компонента должен иметь мета-таблицу, содержащую описание полей и методов класса. Экземпляр компонент должен инициализироваться методом *initWith,* принимающим в качестве параметра структуру с описанием. Экземпляр компонента должен сериализироваться методом *saveTo* в ту же структуру, которая используется для инициализации. Структура описания также должна иметь мета-таблицу.

Мета-таблицы классов создаются в ручном режиме с помощью макросов. Описание метатаблицы состоит из двух частей: заголовка и реализации. Заголовком является макрос *mtMETA_DEC,* помещенный в тело описываемого класса. Реализация помещается в **C++** файл и состоит из следующих макросов: *mtBEGIN(class), mtBASE(baseclass), mtVARIABLE(var), mtMETHOD(foo), mtSIGNAL(event), mtEND*. Применение макросов для определения мета-таблиц показано на рис. 2.

Определение классов

```
class ComponentDef
{
    mtMETA_DEC;
public:
    ECS::Nullable<int> fieldValue;
}


class ComponentClass : public ECS::Component
{
    mtMETA_DEC;
public:
    ComponentClass();
    bool initWith(ComponentDef* def);
    void saveTo(ComponentDef* def);
    void foo(float param);

    int field;

    Signal<void(float, float)> event;
}
```

Определение мета-таблиц

```
mtBEGIN(ComponentDef)
    mtCONSTRUCTOR();
    mtVARIABLE(fieldValue);
mtEND


mtBEGIN(ComponentClass)
    mtBASE(ECS::Component);
    mtCONSTRUCTOR();
    mtVARIABLE(field);
    mtMETHOD(foo);
    mtMETHOD(initWith);
    mtMETHOD(saveTo);
    mtSIGNAL(event);
mtEND
```

Рис. 2. Определение классов компонентов и мета-таблиц

Новые объекты создаются по шаблонам из палитры объектов. Палитра объектов – это набор объектов, агрегирующих структуры описания компонентов. Шаблоны описываются в XML-файлах. Корневой элемент является частью палитры объектов и содержит список элементов *GameObject* с уникальными идентификаторами для каждого шаблона. Несколько файлов могут быть объединены в одну палитру, при этом уникальность идентификаторов должна сохраняться. Вложенные элементы представляют компоненты и название тега должно соответствовать названию класса компонента. Названия атрибутов соответствуют полям структуры описания компонента. Если параметр является структурой, то он может быть описан в виде вложенного XML-элемента. XML-файлы читаются в структуры описаний с помощью алгоритма, использующего мета-данные структур. Поля структур допускают неинициализированные (Nullable) значения. Это позволяет реализовать частичную сериализацию и инициализацию описаний. Инициализация компонентов с помощью структур описаний должна производиться вручную.

Взаимодействие со сторонними библиотеками. Некоторые работы, рассмотренные в аналитической части статьи, представляют подсистемы приложения как отдельные компоненты, мы же предлагаем использовать несколько иной подход. Каждый объект в игре – сборка (Assembly) компонентов, реализующих определенный аспект поведения объекта. Компоненты могут быть объединены в модули, связанные со сторонними библиотеками. Таким образом, каждый компонент является интерфейсом по отношению к подсистеме приложения, как это показано на рис. 2. Это предотвращает некоторые проблемы с производительностью и позволяет использовать библиотеки, которые ничего не «знают» о компонентах и их «устройстве».

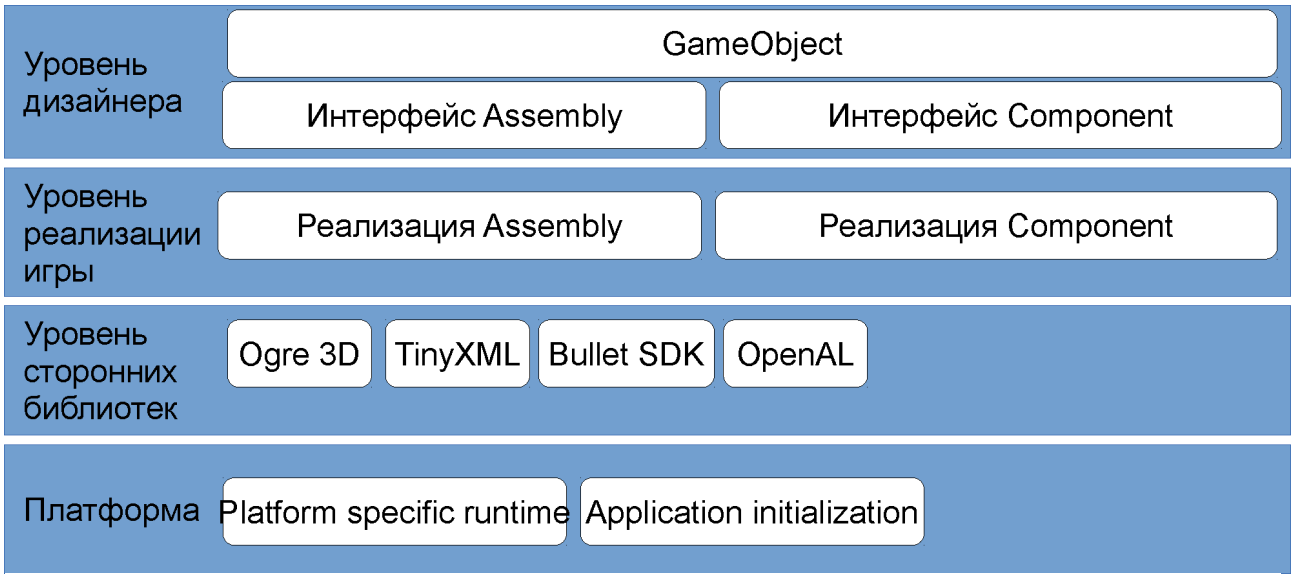

Рис. 3. Уровни приложения

3. Реализация фреймворка.

Система сигналов и слотов. Взаимодействие компонентов осуществляется с помощью системы сигналов и слотов. Эта система реализована аналогично **Boost signals** и **Qt 5**. Подсоединяемые слоты-методы связываются с экземпляром компонента в объект std::function с заданной сигнатурой. Дескриптор соединения сохраняется в защищенных полях компонента для дальнейшего использования. Дескрипторы используются для того, чтобы отключать слоты неактивного или уничтоженного компонента. Библиотека **Qt**, к сожалению, лишилась подобной возможности, начиная с 5-ой версии. Подключение к сигналу анонимных методов и других объектов std::function все еще возможно, но требует ручного управления.

Мета-таблицы. Мета-информация о классах помещается в объекты типа *MetaClass*. Мета-класс содержит указатель на таблицу базового класса, массивы указателей на описания переменных, сигналов и функций. Макрос *mtMETA_DEC* определяет в описании класса два метода (статический и полиморфный), возвращающие экземпляр таблицы *MetaClass*. Макросы *mtBEGIN* и *mtEND* описывают реализацию этих методов и определяют подкласс мета-таблицы целевого класса в анонимном пространстве имен. Макросы *mtVARIABLE*, *mtMETHOD*, и *mtSIGNAL* определяют конструктор создаваемого класса, заполняя списки переменных и методов.

MetaVariable хранит строковый и числовой идентификаторы типа переменной, а также ссылку на поле класса. *MetaVariable* имеет абстрактные методы для чтения и установки значения у экземпляра класса. Оба метода принимают два параметра типа *void** – один для экземпляра объекта, другой – для указателя на значение. Для реализации используется шаблонный класс *impl::MetaVariable<T,F>*, параметризируемый классом, которому принадлежит поле, и типом поля.

Аналогично мета-таблицам переменных, мета-таблицы для функций представлены абстрактным интерфейсом и шаблонной реализацией. Для каждой арности определена своя специализация и в текущей версии фреймворка максимальная арность равна двум. Это ограничение может быть снято при помощи использования шаблонов с переменным количество аргументов. Мета-таблицы методов позволяют вызвать метод с использованием массива параметров, каждый из которых передается через указатель *void**.

3. Апробация.

Демонстрационное приложение. Разработанный фреймворк был протестирован на тестовом игровом приложении, которое разрабатывалось для исследования алгоритмов ИИ в играх. Репозиторий проекта доступен по ссылке [1]. В демо-приложении игроки, управляемые ИИ (юниты) собирали предметы, случайно распределенные по состоящей из квадратных клеток карте (тайлов). Юниты могли перемещаться в 8-ми направлениях, отстоящих друг от друга на 45^0. Каждый юнит имел радиус видимости и мог «видеть» предметы только внутри него. Для того чтобы «подобрать» предмет, юнит должен находиться на соседней с ним клетке.

Для реализации демонстрационного приложения были разработаны следующие компоненты.

1. Transform хранит координаты, поворот и масштаб объекта.

2. SceneNode представляет узел графа сцены библиотеки Cocos2d-x. Этот компонент является абстрактным. Интерфейс *SceneNode* реализуется в классах *SpriteNode* и *LandscapeNode*. *SpriteNode* используется для визуализации статических объектов, предметов и юнитов, *LandscapeNode* для визуализации тайлового ландшафта.

3. MapNavGrid представляет собой карту путей на тайловом ландшафте.

4. MapObstacle описывает препятствие на карте.

5. Actuator – абстрактный компонент, описывающий привод. Реализация *Vehicle* отвечает за перемещение юнитов по карте. Перемещение планируется с помощью алгоритма A*[6].

6. Sensor – абстрактный компонент, описывающий сенсор агента. Реализуется компонентом *Vision*, позволяющим получить список объектов в поле зрения агента.

7. AgentAI – реализует алгоритм поведения юнита, которое определяется набором правил и состоянием памяти.

8. Inventory – компонент, описывающий предметы в инвентаре юнита.

9. *Item* – компонент, описывающий предмет, брошенный на карте.

Сравнение с классическим ООП приложением. В качестве объекта для сравнения классического (эталонного) и предлагаемого подходов возьмем несложное игровое приложение [2], разработанное по классическим принципам ООП. Критерии сравнения: способ инициализации объектов; количество (объем) кода необходимого для реализации аналогичной функциональности; количество зависимостей между классами; сложность добавления новых видов поведения объектов.

Инициализация моделей в обоих приложениях производится одинаково – с помощью структур описания. В эталонном приложении сериализация и десериализация структур описаний производится вручную – это приводит к необходимости написания большого количества схожего кода для каждого объекта в игре. В эталонном приложении поведение каждого вида объектов контролируется соответствующим алгоритмом. Соответственно для добавления нового вида поведения достаточно добавить новый алгоритм (класс). В разработанном приложении поведение объекта может быть изменено добавлением в объект нового компонента из существующих или изменением свойств существующего в процессе выполнения. В эталонном приложении значительно больше жестких зависимостей. Демонстрационное приложение написано с использованием приблизительно 2000 строк кода (метрика SLOC -P). Для реализации той же самой функциональности в эталонном приложении понадобилось 3200 строк. Это различие относительно невелико в демонстрационном примере, однако оно значительно увеличивается в более сложных задачах. Разработанный фреймворк также позволяет реализовать автоматическую привязку встраиваемых языков (**Lua**) при помощи мета-информации о классах и методах. В эталонном приложении использовались отдельные методы, осуществляющие привязку, которые ссылались на все классы приложения – это сильно увеличивало связанность. Таким образом, разработанный фреймворк, построенный на основе

компонент-ориентированной архитектуры, позволяет уменьшить накладные расходы на создание гибких и масштабируемых приложений.

Таким образом, достоинствами фреймворка являются: ориентация на разработку гибких масштабируемых приложений; поведение игрового объекта может быть определено в процессе выполнения; небольшого количество компонентов достаточно для построения большого количества моделей поведения методом комбинации; зависимости между компонентами легко проследить; большинство компонентов не зависят друг от друга; компоненты также могут быть объединены в динамические модули (dll или so) и применены без перекомпиляции основного приложения и использования заголовочных файлов; фреймворк поддерживает частичную десериализацию, что позволяет применять патчи к данным, сохраненным пользователем.

К недостаткам данного фреймворка можно отнести тот факт, что каждый класс должен определяться дважды: один раз в виде стандартного заголовка и второй раз – в виде таблицы метаинформации. Этот недостаток может быть преодолен с помощью генерации кода перед компиляцией. Также предполагается наличие промежуточных структур для сериализации компонентов. С одной стороны поля этих структур в большинстве своем дублируют поля компонентов, с другой – они позволяют различать «поля для сериализации» и поля, используемые для вычислений.

Итак, в рамках проведенной работы сделано следующее.

1. Разработан фреймворк, позволяющий описывать типы компонентов игровых объектов на языке C++. Фреймворк реализует чтение декларативных описаний компонентов из XML-файлов, а также запись соответствующего описания в файл.

2. Реализованы методы динамического связывания сигналов и слотов компонентов на основе мета-информации о классах, методы копирования и частичной инициализации описаний компонентов.

3. Эффективность применения фреймворка подтверждена в демонстрационном приложении.

4. Для игровых приложений, написанных на языке **C++,** предложено эффективное решение проблемы динамического конструирования объектов в процессе выполнения.

5. В отличие от вариантов решений, описанных в аналитическом разделе настоящей статьи, разработанный инструмент является кросс-платформенным, не использует виртуальную машину и не требует дополнительных зависимостей.

Список литературы

1. Алимов А. А. Entity Component Solution Framework / А. А. Алимов. – 2013. – Режим доступа: http://code.google.com/p/aipractice/ (дата обращения 01.10.2013).

2. Алимов А. А. Isilme SDK / А. А. Алимов. – 2011. – Режим доступа: http://code.google.com/p/aipractice/ (дата обращения 01.10.2013).

3. Катаев А. В. Способы проверки решений заданий по программированию в обучающих системах / А. В. Катаев, О. А. Шабалина, В. А. Камаев // Прикаспийский журнал: управление и высокие технологии. – 2011. – № 3. – С. 19–25.

4. Briot Jean-Pierre. Architectural Design of Component-based Agents: a Behavior-based Approach / Briot Jean-Pierre, Meurisse Thomas, Peschanski Frederic // AAMAS'06 4th Int. Workshop on Programming Multi-Agent Systems. ProMAS'06. – Springer, 2006. – P. 35–49.

5. Buro Michael. On the development of a free RTS game engine / Buro Michael, Furtak Timothy // GameOn'NA Conference. – 2005. – C. 23–27.

6. Dechter Rina. Generalized best-first search strategies and the optimality of A* / Dechter Rina, Pearl Judea // J. ACM. – 1985. – № 3. – P. 505 – 536. Available at: http://doi.acm.org/10.1145/3828.3830 (accessed 01.10.2013).

7. Doherty Michael. A Software Architecture for Games / Doherty Michael // University of the Pacic Department of Computer Science Research and Project Journal (RAPJ). – 2003. – Available at:

http://www1.pacic.edu/ mdoherty/research/GAMES/gamearchitecture.pdf (accessed 01.10.2013).

8. Haller Michael. A generic framework for game development / Haller Michael, Zauner Juergen, Hartmann Werner // ACM SIGGRAPH and Eurographics Campre. – Snowbird, Utah, USA, 2002. – ISBN 3-8322-0241-2.

9. Hessam S. Sarjoughian. A Scalable Component-based Modeling Environment Supporting Model Validation / Hessam S. Sarjoughian // Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC). – 2005.

10. Jing Huang. Reflective Design for Component-Based Distributed Systems with Timing Constraints / Jing Huang, Wei Zhang, Yansheng Lu, Qin Yang // Proceedings of the 2008 32nd Annual IEEE International Computer Software and Applications Conference. COMPSAC '08. – Washington, DC, USA : IEEE Computer Society, 2008. – P. 424–427.

11. Microsoft Corporation. Xaml Object Mapping Specification. – 2009. – Available at: http://msdn.microsoft.com/en-us/library/hh857629.aspx (accessed 01.10.2013).

12. OpenSteer. Steering Behaviors for Autonomous Characters. – Available at: http://opensteer.sourceforge.net/ (accessed 01.10.2013).

13. Unity Technologies. Unity Documentation. – Available at: http://unity3d.com/learn /documentation (accessed 01.10.2013).

References

1. Alimov A. *Entity Component Solution Framework.* 2013. Available at: http://code.google.com/p/aipractice/ (accessed 01.10.2013). (In Russ.)

2. Alimov A. *Isilme SDK.* 2013. Available at: https://code.google.com/p/isilme/ (accessed 01.10.2013). (In Russ.)

3. Kataev A. V., Shabalina O. A., Kamaev V. A. Sposoby proverki resheniy zadaniy po programmirovaniyu v obuchayushchikh sistemakh [Means of verification of task solutions for programming in training systems]. *Prikaspiyskiy zhurnal: upravlenie i vysokie tekhnologii* [Caspian Journal: Management and High Technologies], 2011, no 2, pp. 19–25.

4. Briot Jean-Pierre, Meurisse Thomas, Peschanski Frederic. Architectural Design of Component-based Agents: a Behavior-based Approach. *AAMAS'06 4th Int. Workshop on Programming Multi-Agent Systems. ProMAS'06.* Springer, 2006, pp. 35–49.

5. Buro Michael, Furtak Timothy. On the development of a free RTS game engine. *GameOn'NA Conference*, 2005, pp. 23–27.

6. Dechter Rina, Pearl Judea. Generalized best-first search strategies and the optimality of A*. *J. ACM*, 1985, no. 3, pp. 505–536. Available at: http://doi.acm.org/10.1145/3828.3830 (accessed 01.10.2013).

7. Doherty Michael. A Software Architecture for Games. *University of the Pacic Department of Computer Science Research and Project Journal (RAPJ).* 2003. Available at: http://www1.pacic.edu/ mdoherty/research/GAMES/gamearchitecture.pdf (accessed 01.10.2013).

8. Haller Michael, Zauner Juergen, Hartmann Werner. A generic framework for game development. *ACM SIGGRAPH and Eurographics Campre*. Snowbird, Utah, USA, 2002. ISBN 3-8322-0241-2.

9. Hessam S. Sarjoughian. A Scalable Component-based Modeling Environment Supporting Model Validation. *Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC)*, 2005.

10. Jing Huang, Wei Zhang, Yansheng Lu, Qin Yang // Reflective Design for Component-Based Distributed Systems with Timing Constraints. *Proceedings of the 2008 32nd Annual IEEE International Computer Software and Applications Conference. COMPSAC '08.* Washington, DC, USA, IEEE Computer Society, 2008, pp. 424–427.

11. Microsoft Corporation. Xaml Object Mapping Specification. 2009. Available at: http://msdn.microsoft.com/en-us/library/hh857629.aspx (accessed 01.10.2013).

12. OpenSteer. Steering Behaviors for Autonomous Characters. Available at: http://opensteer. sourceforge.net/ (accessed 01.10.2013).

13. Unity Technologies. Unity Documentation. Available at: http://unity3d.com/learn /documentation (accessed 01.10.2013).