GeekBrains

Паттерны проектирования ПО

Введение. Паттерны Singleton и Proxy

На этом уроке

1. Изучим принципы ООП SOLID. Вспомним принципы разработки KISS, DRY, YAGNI. Рассмотрим классификацию паттернов по книге GOF. Узнаем об UML и диаграмме классов. Рассмотрим диаграмму классов «Заместитель» (Proxy).
2. Разработаем на C++ проект игры SBomber. Рассмотрим UML-диаграмму паттерна «Одиночка» и его реализацию на C++. Применим паттерн «Одиночка» в нашем игровом проекте. Изучим паттерн «Заместитель».

Оглавление

Введение

Шаблон проектирования или **паттерн** (design pattern) в разработке программного обеспечения — повторяемый архитектурный приём, который решает определённую проблему проектирования [1].

В отличие от готовых функций или библиотек, паттерн нельзя взять и скопировать в программу. Паттерн — это не конкретный код, а общая концепция решения той или иной проблемы. Такую концепцию ещё нужно адаптировать для нужд конкретного проекта [2].

Важно не путать паттерны с алгоритмами! Оба понятия описывают типовые решения проблем. Но алгоритм — это чёткий набор действий, а паттерн — высокоуровневое описание решения, которое можно по-разному реализовать в разных программах.

Паттерны часто изображают с помощью диаграмм UML (Unified Modeling Language). Диаграммы на этом языке будем использовать и мы. Базовые знания UML, особенно диаграммы классов, полезны и при работе с кодом и для объяснении своих идей коллегам-рограммистам.

Признаки плохого кода

Есть несколько признаков того, что код нужно улучшать. Например, он стабильно работает, но не предусматривает возможности будущих изменений.

Посмотрите на список признаков плохого кода. Чем больше пунктов справедливы для вашего проекта, тем сильнее код нуждается в рефакторинге [4].

- **Закрепощенность** — необходимость для изменения одного участка кода править другой, третий, пятый и так далее.
- **Неустойчивость** — ситуация, когда при изменении одного участка кода внезапно перестаёт работать другой, казалось бы, напрямую не связанный участок.
- **Неподвижность** — невозможность повторного использования написанного ранее кода.

- **Неоправданная сложность** — использование сложных синтаксических конструкций без реальной выгоды.
- **Плохая читаемость** — некрасивый и сложный для понимания листинг программы.

Рефакторинг (англ. refactoring) — это переработка исходного кода программного обеспечения с целью сделать его более простым и понятным. В ходе рефакторинга мы не меняем поведение программы (её алгоритмов), не исправляем ошибок и не добавляем новых возможностей. Но мы делаем код более понятным и удобочитаемым.

Использование принципов SOLID помогает создать систему, которую легко поддерживать и расширять в течение долгого времени. Принципы SOLID также могут применяться во время работы над существующим программным обеспечением. Например, для удаления «дурно пахнущего кода».

Принципы объектно-ориентированного программирования SOLID

SOLID (сокр. от англ. single responsibility, open–closed, Liskov substitution, interface segregation и dependency inversion) в программировании — мнемонический акроним, введённый Майклом Фэзерсом (Michael Feathers) для первых пяти принципов объектно-ориентированного программирования и проектирования [3], предложенных Робертом Мартином в начале 2000-х.

В переводе с английского, слово solid означает «твёрдый», «прочный», «надёжный», «цельный». Такими же свойствами обладают и программы, написанные с использованием этих принципов. Это не волшебная палочка, которая гарантирует выдающийся результат, но следование принципам SOLID помогает создавать хороший код [4].

5 принципов SOLID:

S — принцип единственной ответственности (single responsibility principle). Для каждого класса должно быть определено единственное назначение. Все ресурсы

для его осуществления должны быть инкапсулированы в этот класс и подчинены только этой задаче.

Пример. Программа позволяет писать логи в файл, работать с БД и обмениваться сообщениями по сети. Не стоит всё это реализовывать в одном классе. Такой класс будет перегруженным, противоречивым — поддерживать и развивать его будет сложно. Лучше использовать как минимум три класса, у каждого из которых будет только одно назначение.

O — принцип открытости/закрытости (open–closed principle): «Программные сущности <...> должны быть открыты для расширения, но закрыты для модификации».

Пояснение. Под модификацией здесь понимают изменение кода существующих классов, а под расширением — добавление новой функциональности. В общем, реализация новых возможностей не должна требовать правок в уже существующем коде класса. Изменение готового кода — это риск сломать существующую архитектуру проекта и пропустить баги. Поэтому лучше не трогать уже протестированный и в целом надежный код, но наращивать функциональность в рамках существующей архитектуры.

L — принцип подстановки «Лисков» (Liskov substitution principle): «Объекты в программе должны быть заменяемыми на экземпляры их подтипов без изменения правильности выполнения программы». Производный класс и родительский класс должны быть взаимозаменяемы.

Пример. Если программный контейнер работает с объектом «Самолёт», значит, он должен так же работать с объектами-наследниками: «Пассажирский самолет» или «Грузовой самолет».

I — Принцип разделения интерфейса (interface segregation principle): «Много интерфейсов, специально предназначенных для клиента, лучше, чем один интерфейс общего назначения».

Пример. Вспомним программу, которая иллюстрировала принцип единственной ответственности. Мы не хотели реализовывать все в одном классе. Сходный принцип верен и для интерфейсов. Не стоит создавать один универсальный

интерфейс для работы с логом БД и чатом. Пользователю такого интерфейса пришлось бы давать много лишних прав, что повысило бы риск неправильного взаимодействия с системом. К тому же, универсальный интерфейс перегружен и противоречив с точки зрения назначения и использования. Лучше создать 3 специализированных интерфейса.

D — принцип инверсии зависимостей (dependency inversion principle): «Зависимость на Абстракциях. Нет зависимости на что-то конкретное».

Это значит, что, во-первых, классы высокого уровня не должны зависеть от низкоуровневых классов. Все они должны зависеть от абстракций. Во-вторых, абстракции не должны зависеть от деталей, но детали должны зависеть от абстракций. Классы высокого уровня реализуют бизнес-правила или логику в приложении. Низкоуровневые классы занимаются более конкретными операциями, такими как запись информации в базу данных или передача сообщений в операционную систему и службы.

Когда класс явно знает о дизайне и реализации другого класса, возникает риск того, что изменения в одном классе нарушат работу другого класса. Поэтому важно держать высокоуровневые и низкоуровневые классы слабо связанными. Для этого нужно сделать их зависимыми от абстракций, а не друг от друга.

Следование этим принципам поможет вам создать качественный код, который будет легче поддерживать. Это не панацея, но это упростит жизнь вам и возможно другим программистам, которые будут работать с вашим кодом.

Принципы DRY, KISS, YAGNI

За этими аббревиатурами скрываются известные принципы разработки, которые помогают создавать более качественное ПО:

DRY — (англ. Don’t repeat yourself) — не повторяйся. Принцип разработки ПО, согласно которому нужно стремиться к минимальной повторяемости информации, особенно в сложных системах. Любая информация в рамках информационной системы должна иметь единственное, непротиворечивое и авторитетное представление. Нарушения принципа DRY *(англ. сухой)* называют WET (англ.

мокрый) — «Write Everything Twice» — буквально «Пиши всё по два раза» [5]. Это своего рода антипаттерн, которого следует избегать.

Пример. Дублирование данных и алгоритмов вредно не только в программировании, но и в работе с базами данных. Например, если мы храним паспортные данные людей в двух разных таблицах БД, то позже при обновлении одной таблицы (кто-то получил новый паспорт) можем забыть внести изменения в другую. Это создаст противоречие в информационной системе и приведёт к сбою её работы.

KISS — (англ. Keep it simple) — не усложняй. Этот принцип проектирования появился в ВМС США. Согласно ему, большинство систем работают лучше, если остаются простыми, а не усложняются. Поэтому в области проектирования простота должна быть одной из ключевых целей, и следует избегать лишней сложности [6].

Пример. Допустим, для решения задачи есть два алгоритма: простой и сложный, изощрённый. Если нас устраивает скорость и сложность простого алгоритма, лучше использовать его. Его мы скорее реализуем правильно и сможем без проблем поддерживать в будущем.

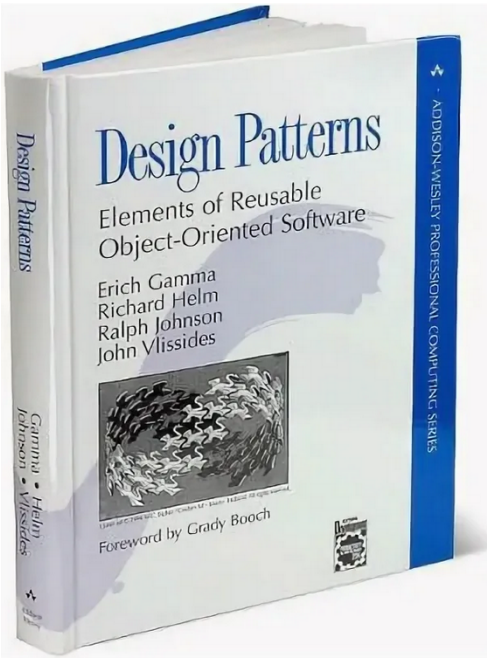
YAGNI — (англ. You aren't gonna need it) — вам это не понадобится. Принцип проектирования ПО, который состоит в отказе от избыточной функциональности [7].

Пример. Нам надо разработать программную систему с поддержкой архива ZIP. Мы хотели как лучше и добавили ещё поддержку форматов RAR и ACE. Это заняло дополнительное оплачиваемое рабочее время, но лишние форматы оказались не нужны пользователям. Поддержка лишней функциональности только увеличила объём и сложность информационной системы. Поэтому при добавлении возможности всегда лучше придерживаться ТЗ.

Классификация паттернов по книге GOF

В 1994 году вышла книга «Приёмы объектно-ориентированного проектирования. Паттерны проектирования» (Design Patterns: Elements of Reusable Object-Oriented Software). Авторы книги известны как «Банда четырёх» (англ. Gang of Four): Эрих

Гамма, Ричард Хелм, Ральф Джонсон и Джон Влиссидес. Предисловие написал создатель унифицированного языка моделирования UML Гради Буч [8].



Книга состоит из двух частей. Первые две главы рассказывают о возможностях и недостатках объектно-ориентированного программирования. Дальше читателем представлены 23 классических шаблона проектирования. Примеры в книге написаны на C++ и Smalltalk.

Все описанные в книге шаблоны проектирования (паттерны) можно разделить на 3 вида: порождающие, структурные и поведенческие.

Порождающие шаблоны проектирования: Abstract Factory — «Абстрактная фабрика», Builder — «Строитель», Factory Method — «Фабричный метод», Prototype — «Прототип» и Singleton — «Одиночка».

Структурные шаблоны проектирования: Adapter — «Адаптер», Bridge — «Мост», Composite — «Компоновщик», Decorator — «Декоратор», Facade — «Фасад», Flyweight — «Приспособленец», Proxy — «Заместитель».

Поведенческие шаблоны проектирования: Chain of responsibility — «Цепочка обязанностей», Command — «Команда», Interpreter — «Интерпретатор», Iterator — «Итератор», Mediator — «Посредник», Memento — «Хранитель», Observer —

«Наблюдатель», State — «Состояние», Strategy — «Стратегия», Template method — «Шаблонный метод», Visitor — «Посетитель».

Некоторые известные архитектурные паттерны в книге не рассматривались. Например, **RAII** (Resource Acquisition Is Initialization) — получение ресурса есть инициализация) и **PIMPL** (Pointer to Implementation) – указатель на реализацию. Их мы также рассмотрим в этом курсе.

Диаграмма классов в UML

UML (англ. Unified Modeling Language) — унифицированный язык моделирования. Используется для проектирования систем (в основном программных), представления организационных структур и моделирования бизнес-процессов [9]. С помощью графических обозначений языка можно создавать абстрактную модель, которую называют UML-моделью.

UML не является языком программирования, но на основе UML-моделей можно генерировать код.

UML позволяет создавать 12 видов диаграмм — структурных и поведенческих, — но в этом курсе мы будем использовать только *диаграмму классов*.

Для общего ознакомления перечислим все типы диаграмм.

Структурные диаграммы:

1. классов,
2. компонентов,
3. составной структуры,
4. размещения,
5. объектов,
6. артефактов.

Поведенческие диаграммы:

1. вариантов использования,
2. взаимодействия,
3. последовательности,
4. коммуникации,

5. состояний,
6. деятельности.

Диаграмма классов (англ. class diagram) — структурная UML-диаграмма, которая показывает общую структуру иерархии классов системы, их кооперaций, атрибутов (полей), методов, интерфейсов и взаимосвязей между ними. Широко применяется для задач документирования и визуализации, также прямого и обратного проектирования [10].

Диаграмму классов создают, чтобы графически представить статическую структуру декларативных элементов системы, например, классов, типов. Диаграмма содержит элементы поведения (например, операции), но для отражения их динамики нужны диаграммы других видов: коммуникации, состояний.

Ради удобства восприятия диаграмму классов можно дополнить представлением пакетов, в том числе вложенных. UML-пакет (англ. package) предназначен для группировки множества структурных, поведенческих и других сущностей (алгоритмов, классов, функций) в единое целое. На языке программирования C++ такой пакет можно реализовать как библиотеку или пространство имен. На диаграмме классов его обычно изображают в виде папки.

Чтобы представить сущности из реального мира, разработчику нужно отразить их текущее состояние, поведение и взаимные отношения. На каждом этапе сущность абстрагируется от маловажных деталей и концепций, которые не относятся к реальности: производительность, инкапсуляция, видимость и др.. Классы обычно относят к разным уровням. Три основных: аналитический уровень, уровень проектирования и уровень реализации.

На уровне анализа класс содержит только общие контуры системы и работает как логическая концепция предметной области или программного продукта.

На уровне проектирования класс отражает решения, связанные с распределением информации и планируемой функциональности. Он объединяет в себе сведения о состоянии и операциях.

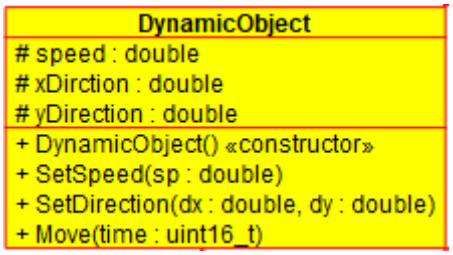
На уровне реализации класс дорабатывается до вида, который удобнее всего воплотить в выбранной среде разработки. При этом опускаются общие свойства, не применимые на выбранном языке программирования.

С переходом на каждый следующий уровень класс становится конкретнее: обрастает деталями и теряет неиспользуемые в реализации свойства.

Диаграмма классов — ключевой элемент объектно-ориентированного моделирования.

На диаграмме классы представлены в рамках и содержат:

- Имя класса, выровненное по центру и написанное полужирным шрифтом. Имена классов начинаются с заглавной буквы. Если класс абстрактный, его имя пишется полужирным курсивом.
- Поля (атрибуты) класса, выровненные по левому краю. Вертикально расположены по середине. Пишутся с маленькой буквы.
- Методы класса. Тоже выровнены по левому краю.



Чтобы любые члены класса — атрибуты или методы — были видимы, обозначения надо разместить перед именем участника:

+	Публичный (Public)
-	Приватный (Private)
#	Защищённый (Protected)
/	Производный (Derived) (может быть совмещён с другими)
~	Пакет (Package)

« / » — этот атрибут является производным от другого атрибута того же класса. Значение производного атрибута (derived element) для отдельных объектов можно вычислить с использованию значений других атрибутов того же объекта.

« ~ » — атрибут с областью видимости типа «Пакет» (package). Недоступен или не виден для всех классов за пределами пакета, в котором определен класс-владелец атрибута.

Взаимосвязи объектов и классов

Взаимосвязь — это тип логических отношений между сущностями. В UML предусмотрены следующие виды отношений на диаграммах классов и объектов [10]:

Тип отношения	UML-синтаксис		Краткая семантика
	источник	цель	
Зависимость			Исходный элемент зависит от целевого элемента и изменение последнего может повлиять на первый.
Ассоциация			Описание набора связей между объектами.
Агрегация			Целевой элемент является частью исходного элемента.
Композиция			Строгая (более ограниченная) форма агрегирования.
Включение			Исходный элемент содержит целевой элемент.
Обобщение			Исходный элемент является специализацией более обобщенного целевого элемента и может замещать его.
Реализация			Исходный элемент гарантированно выполняет контракт, определенный целевым элементом.

Зависимость (dependency) — это слабая форма отношений пользования, при которых изменение в спецификации одного элемента влечёт изменение другого, причём обратное необязательно. Зависимость возникает, когда объект служит, например, параметром или локальной переменной.

Графически зависимость представляют штриховой стрелкой, идущей от зависимого элемента к тому, от которого он зависит. Зависимость может быть между экземплярами, классами или экземпляром и классом.

Ассоциация показывает, что объекты одной сущности (класса) связаны с объектами другой сущности таким образом, что можно перемещаться от объектов одного класса к другому. Это общий случай композиции и агрегации.

Пример. Классы «Человек» и «Школа» имеют ассоциацию, потому что человек может учиться в школе. Такой ассоциации можно присвоить имя «учится в».

Двойные ассоциации изображают в виде линии без стрелок, соединяющей два классовых блока. Ассоциации более высокой степени имеют больше двух концов и изображаются линиями, один конец которых идёт к классовому блоку, а другой — к общему ромбу. В представление однонаправленной ассоциации добавляют стрелку, которая указывает на направление ассоциации.

Ассоциации можно давать имя, а на концах представляющей её линии — подписывать роли, принадлежности, индикаторы, мультипликаторы, видимости или другие свойства.

Агрегация — разновидность ассоциации при отношении между целым и его частями. Как тип ассоциации, агрегация может быть именованной. Одно отношение агрегации может включает максимум 2 класса: контейнер и содержимое.

Агрегация встречается, когда один класс является коллекцией или контейнером других. Причём по умолчанию агрегацией называют *агрегацию по ссылке*, при которой время существования содержащихся классов не зависит от времени существования контейнера. Если контейнер будет уничтожен, его содержимое — нет.

Графически агрегацию представляют пустым ромбом на блоке класса и линией, идущей от этого ромба к содержащемуся классу.

Композиция — более строгий вариант агрегации, также известный как «агрегация по значению».

В композиции экземпляры содержащихся классов жёстко зависят от экземпляров класса контейнера. При уничтожении контейнера его содержимое также будет уничтожено. Графически композию представляют как агрегацию с закрашенным ромбом.

Различия между композицией и агрегацией

Комната является частью дома, следовательно, эти объекты связаны по принципу композиции: комната без дома существовать не может. А вот мебель — не всегда часть дома, но может в нём находиться, поэтому связь этих сущностей будет агрегацией.

Обобщение (Generalization) — показывает, что один из двух связанных классов является частной формой другого. На практике это значит, что любой экземпляр подтипа является также экземпляром надтипа. Пример: животные — супертип млекопитающих, которые, в свою очередь, — супертип приматов, и так далее. Эту взаимосвязь легче всего описать фразой «А — это Б»: приматы — это млекопитающие, млекопитающие — это животные.

Графически обобщение представляют в виде линии с пустым треугольником у супертипа. Обобщение также известно как «наследование» или отношение «является» (is a...).

Реализация — отношение между двумя элементами, из которых один (клиент) реализует поведение, заданное другим (поставщиком). Реализация — отношение целого и части. Графически реализацию обозначают как наследование, но с пунктирной линией. Поставщиком обычно выступает абстрактный класс или класс-интерфейс.

Примечание: для создания UML-диаграмм классов есть бесплатные программы [11]. Некоторые из них поддерживает генерацию кода на C++ и реверс-инжиниринг, что очень удобно. Например, чтобы создать диаграмму классов вашего проекта, достаточно «натравить» программу на директорию с исходным кодом.

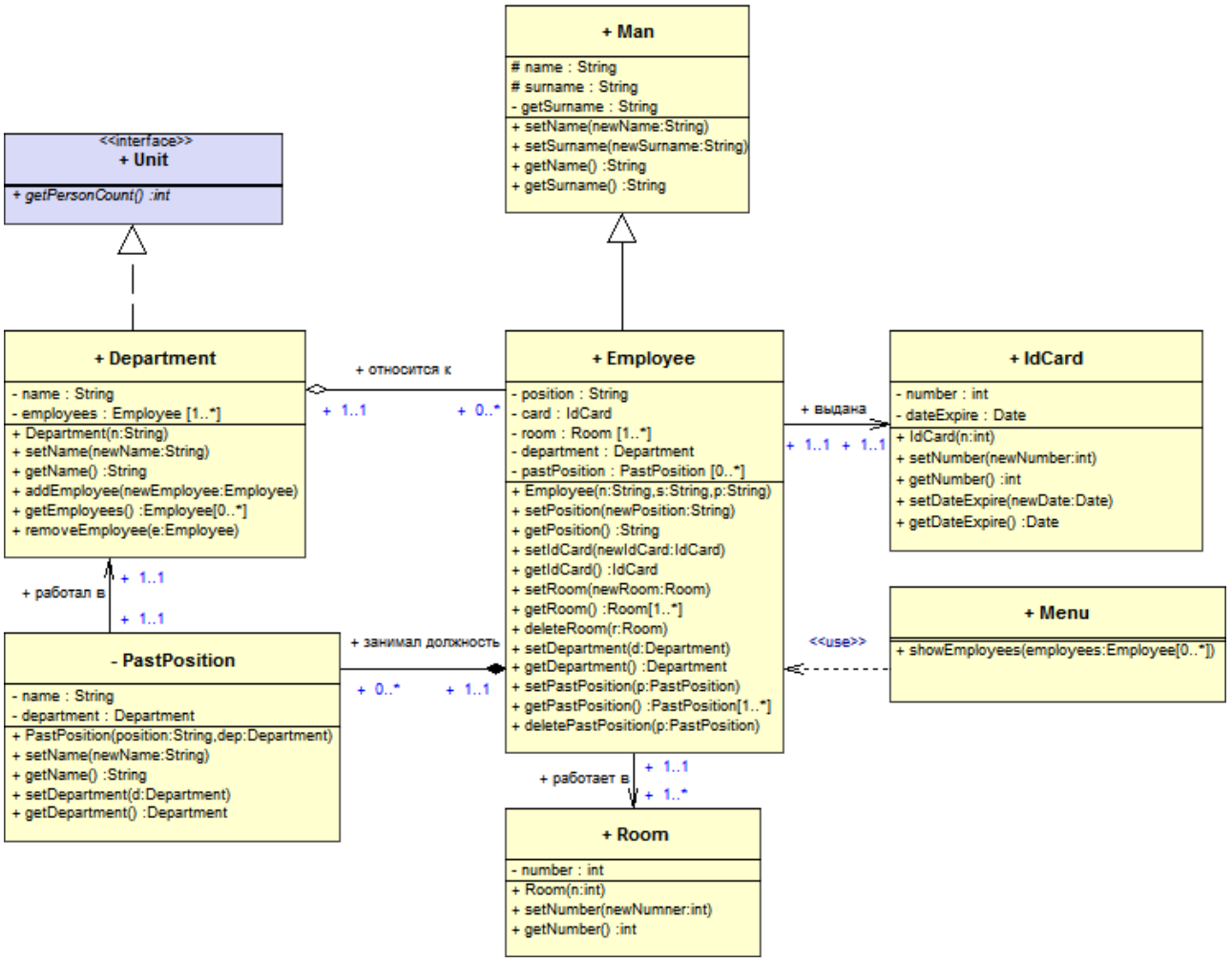
Пример диаграммы классов

Рассмотрим пример диаграммы классов на UML (см ниже).

Класс «Сотрудник» (Employee) — наследник класса человек (Man). Поля «Сотрудника» — номер ID-карты и рабочего кабинета, а также предыдущие должности (PastPosition).

Класс «Отдел» (Department) содержит список (массив) сотрудников отдела. Класс «Предыдущая позиция» также содержит информацию об отделе, где сотрудник работал в прошлом.

Класс «Меню» при отображении сотрудников получает их список в виде массива.



Обзор проекта игры SBomber

Название SBomber означает Simple Bomber — «простой бомбардировщик», что отражает суть игры. Возможно, проект не идеален в плане архитектуры и быстродействия, но он достаточно сложен и разнообразен, чтобы на его примере нам было интересно опробовать разные паттерны проектирования.

Рассмотрим функцию main (главный цикл анимации):

```
int main(void)
{
	MyTools::OpenLogFile("log.txt");

	SBomber game;

	do {
		game.TimeStart();

		if (_kbhit())
		{
		game.ProcessKBHit();
		}

		MyTools::ClrScr();

		game.DrawFrame();
		game.MoveObjects();
		game.CheckObjects();

		game.TimeFinish();

	} while (!game.GetExitFlag());

	MyTools::CloseLogFile();
	return 0;
}
```

В главном цикле анимации вызываются методы объекта класса SBomber:

- Засекаем время рисования каждого кадра в миллисекундах.
- Проверяем ввод с клавиатуры. Если нажатие было, для его обработки вызываем функцию `ProcessKBHit`.
- Очищаем кадр.
- Рисуем его заново.
- Перемещаем динамические объекты.

- Проверяем столкновение объектов.
- Для выхода из цикла проверяем значение флага через функцию-геттер GetExitFlag.
- По ходу выполнения программы логируем разные события в текстовый файл.

Главный класс игры:

```
class SBomber
{
public:

	SBomber();
	~SBomber();

	inline bool GetExitFlag() const { return exitFlag; }

	void ProcessKBHit();
	void TimeStart();
	void TimeFinish();

	void DrawFrame();
	void MoveObjects();
	void CheckObjects();

private:

	void CheckPlaneAndLevelGUI();
	void CheckBombsAndGround();
	void __fastcall CheckDestoyableObjects(Bomb* pBomb);

	void __fastcall DeleteDynamicObj(DynamicObject * pBomb);
	void __fastcall DeleteStaticObj(GameObject* pObj);

	Ground * FindGround() const;
	Plane * FindPlane() const;
	LevelGUI * FindLevelGUI() const;
	std::vector<DestroyableGroundObject*> FindDestoyableGroundObjects() const;
	std::vector<Bomb*> FindAllBombs() const;

	void DropBomb();

	std::vector<DynamicObject*> vecDynamicObj;
	std::vector<GameObject*> vecStaticObj;

	bool exitFlag;

	uint64_t startTime, finishTime, passedTime;
	uint16_t bombsNumber, deltaTime, fps;
	int16_t score;
};
```

У класса SBomber есть 7 публичных функций-методов, доступных для вызова:

- геттер для получения флага выхода из цикла анимации GetExitFlag;
- обработчик нажатий на клавиатуре ProcessKBHit;
- функции для получения времени рисования кадра (TimeStart и TimeFinish);
- функция рисования кадра (всех игровых объектов) DrawFrame;
- функция перемещения динамических объектов — MoveObjects;
- функция-метод для проверки столкновений объектов — CheckObjects.

Группы внутренних приватных функций методов

Первая группа: функции Check для проверки пересечения динамических объектов.

Вторая группа: две функции Delete для удаления ненужных динамических и статических объектов.

Третья группа: функции Find для поиска нужных объектов по массивам. Возвращают массив указателей на найденные объекты.

Есть ещё функция броска бомбы — DropBomb.

Обратите внимание! Соглашение о вызовах __fastcall, которое можно видеть перед началом некоторых функций-методов, рекомендует компилятору передавать параметры функции через регистры микропроцессора вместо стека. Такой вызов должен быть быстрее. Если у функции нет параметров или их очень много, использовать этот спецификатор нет смысла.

Главный класс игры SBomber содержит объекты, из которых состоит игра. Это массивы Static- и Dynamic-объектов.

Статические объекты имеют только свои координаты и умеют рисовать (виртуальная функция Draw) себя.

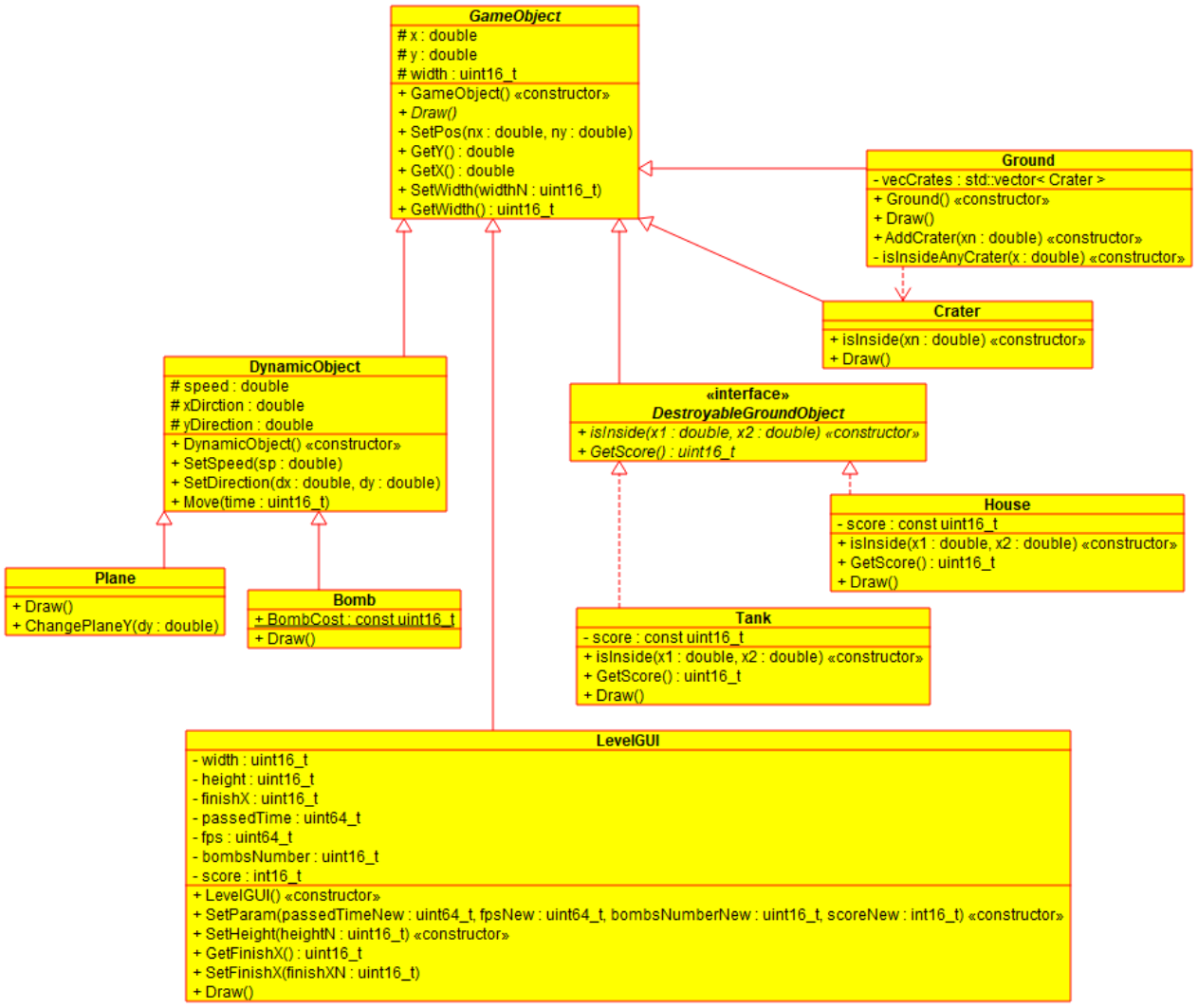
Динамические — умеют ещё и перемещаться с помощью виртуальной функции Move. В этом их основное отличие.

К статическим объектам в игре относят Ground, Crater, LevelGUI, Tank и House. Объект класса Ground может содержать воронки от бомб, то есть объекты класса Crater.

К динамическим относят Plane и Bomb. У динамических также есть скорость и вектор движения.

Ещё есть группа наземных разрушаемых объектов — класс DestroyableGroundObject. Сюда относятся Tank и House. Объект LevelGUI позволяет рисовать интерфейс: рамку, внутри которой происходит анимация и сама игра, а также игровую статистику, такую как время миссии, количество кадров в секунду, количество заработанных очков, доступное количество бомб у самолета.

Вот иерархия объектов игры, представленная в виде диаграммы классов:



На диаграмме показаны такие типы отношений, как наследование, реализация и зависимость.

А вот как выглядит игра SBomber после запуска:

Исходный код игры SBomber вы можете скачать в виде архива. Проект разработан в Microsoft Visual Studio (Community version). Если выберете ту же IDE, вы сможете сразу открыть проект без необходимости что-то настраивать и адаптировать.

Управление:

- ☐ Курсорные клавиши — управление самолетом
- ☐ B (латинская) — бросить бомбу.
- ☐ ESC — выход.

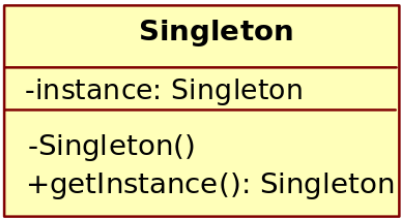
Каждая брошенная бомба забирает 10 очков. Подбитый танк дает 30 очков, а домик — 40.

Паттерн «Одиночка» (Singleton)

Одиночка (англ. Singleton) — порождающий шаблон проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет глобальную точку доступа к этому экземпляру. Можно сказать, что «Одиночка» — своеобразный аналог глобальной переменной, доступность которой является одновременно её преимуществом и недостатком. Это самый известный шаблон проектирования из

рассмотренных в книге GoF. На собеседовании вас могут спросить, как реализовать этот паттерн на языке C++.

Вот упрощённая UML-диаграмма паттерна:



Знак «-» на диаграмме означает закрытый — private — к методу или переменной. А «+» говорит о доступности снаружи, то есть символизирует спецификатор public. На диаграмме видим приватный экземпляр класса Singleton и приватный конструктор. Это значит, что снаружи новый объект класса не сконструировать. Ещё видим публичную — обычно статическую — функцию getInstance, которая возвращает ссылку на приватный экземпляр.

«Одиночки» могут содержать дополнительные прикладные функции-методы типа getData, setData и другие, необходимые для выполнения прикладной логики, ради которой и создавался класс.

Преимущества паттерна «Одиночка» [13]:

- Гарантирует наличие единственного экземпляра класса.
- Предоставляет к нему глобальную точку доступа.
- Реализует отложенную инициализацию объекта-одиночки.

Недостатки:

- Нарушает принцип единственной ответственности класса.
- Маскирует плохой дизайн.
- Имеет проблемы с многопоточностью.
- Требует постоянного создания Mock-объектов при юнит-тестировании.

Ниже приведена одна из возможных реализаций паттерна «Одиночка» на C++, известная как «синглтон Майерса». В ней «Одиночка» представляет собой статический локальный объект [12]. Важный момент: конструктор класса объявлен

как private, что позволяет предотвратить создание экземпляров класса за пределами его реализации. Закрытыми также объявлены конструктор копирования и оператор присваивания. Последние следует объявлять, но не определять, чтобы, в случае их случайного вызова из кода, получить легко обнаруживаемую ошибку компоновки. Отметим также, что приведенный пример не является потокобезопасным в C++03. Для работы с классом из нескольких потоков нужно защитить переменную theSingleInstance от одновременного доступа, например, с помощью мьютекса или критической секции. Впрочем, в C++11 «синглтон Майерса» потокобезопасен безо всяких блокировок.

```
class OnlyOne
{
public:
    static OnlyOne& getInstance()
    {
        static OnlyOne theSingleInstance;
        return theSingleInstance;
    }
private:
    OnlyOne() { }
    OnlyOne(const OnlyOne& root) = delete;
    OnlyOne& operator=(const OnlyOne&) = delete;
};
```

Применение паттерна «Одиночка» в проекте SBomber

В проекте SBomber есть единица трансляции MyTools.cpp, которая содержит две группы вспомогательных в проекте функций. Рассмотрим первую группу:

```
void ClrScr();
void __fastcall GotoXY(double x, double y);
uint16_t GetMaxX();
uint16_t GetMaxY();
void SetColor(ConsoleColor color);
```

Перед нами функции работы с экраном: очистка, перевод курсора по заданным координатам, получение максимального значения координат x и y, задание цвета текста. Но эти алгоритмы реализованы в процедурном стиле программирования, не

в ООП. Давайте отрефакторим их с применением паттерна «Одиночка» и объединим их функциональность у него «под капотом».

Пример реализации класса ScreenSingletone:

```
#pragma once

#include <stdint.h>

// Палитра цветов от 0 до 15
enum ConsoleColor
{
   CC_Black = 0,
   CC_Blue,
   CC_Green,
   CC_Cyan,
   CC_Red,
   CC_Magenta,
   CC_Brown,
   CC_LightGray,
   CC_DarkGray,
   CC_LightBlue,
   CC_LightGreen,
   CC_LightCyan,
   CC_LightRed,
   CC_LightMagenta,
   CC_Yellow,
   CC_White
};

class ScreenSingleton
{
public:

   static ScreenSingleton& getInstance()
   {
      static ScreenSingleton theInstance;
      return theInstance;
   }

   void ClrScr();
   void __fastcall GotoXY(double x, double y);
   uint16_t GetMaxX();
```

```
    uint16_t GetMaxY();
    void SetColor(ConsoleColor color);

private:
    ScreenSingleton() { }
    ScreenSingleton(const ScreenSingleton& root) = delete;
    ScreenSingleton& operator=(const ScreenSingleton&) = delete;
};
```

Теперь нужно заменить код в местах вызова функций работы с экраном: в функции main, виртуальных функциях Draw, модуле SBomber.cpp и др.).

Пример:

```
ClrScr();
GotoXY(x, y);
```

Меняем на:

```
ScreenSingleton::getInstance().ClrScr();
ScreenSingleton::getInstance().GotoXY(x, y);
```

Структурный паттерн «Заместитель» (Proxy)

«Заместитель» (англ. Proxy) — структурный шаблон проектирования, который предоставляет объект для контроля доступа к другому объекту и при этом перехватывает все вызовы, то есть выступает контейнером [14].

Плюсы паттерна [15]

Ленивая инициализация (виртуальный прокси). Позволяет создавать «тяжёлый» объект, грузящий данные из файловой системы или базы данных, только при первом обращении к функции.

Защита доступа (защищающий прокси). Позволяет защищать объект от неавторизованного доступа при наличии разных типов пользователей в программе. Например, если ваши объекты — важная часть операционной системы, а пользователи — сторонние программы (хорошие или вредоносные).

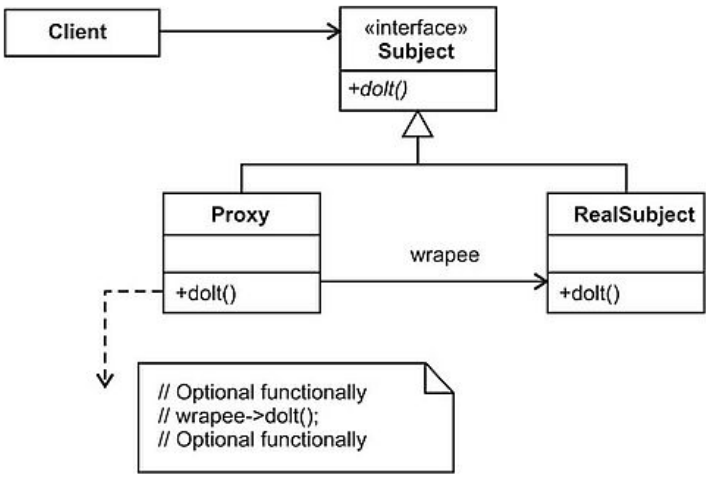
Локальный запуск сервиса (удалённый прокси). Когда настоящий сервисный объект находится на удалённом сервере.

Логирование запросов (логирующий прокси). Когда требуется хранить историю обращений к сервисному объекту.

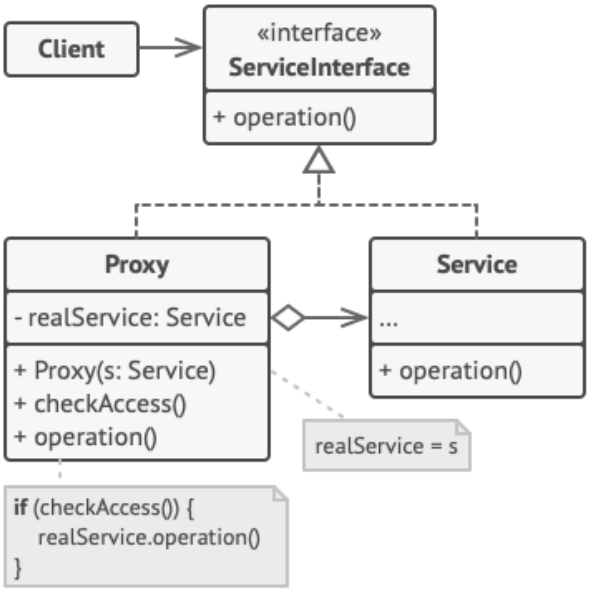
Кеширование объектов («умная» ссылка). Когда нужно кешировать результаты запросов клиентов и управлять их жизненным циклом.

Минус паттерна «Заместитель» — резкое увеличение времени отклика.

Рассмотрим несколько примеров диаграммы классов UML для паттерна «Заместитель».



Видим интерфейс и две его реализации: Proxy и RealSubject. Вызовы функции dolt у Proxy на самом деле передаются объекту RealSubject, который содержится в Proxy. То есть «Заместитель» оборачивает наш реальный объект. Клиент может взаимодействовать с Proxy через интерфейс: это добавит гибкости и будет неотличимо от взаимодействия с RealSubject.



Вторая диаграмма классов немного сложнее и позволяет задавать «Заместителю» — через его конструктор — разные объекты Service, которые он обернёт, что даст нам большую гибкость и удобство. Также видим проверку доступа к сервису с помощью функции checkAccess.

Внимание! Сохраните исходный код проекта

После применения паттерна Singletone в проекте SBomber (ScreenSingleton и FileLoggerSingleton) сделайте копию исходного кода проекта, чтобы использовать её в будущих семи уроках.

Чтобы не слишком усложнять проект, мы не будем копить в нём примененные ранее паттерны, за исключением «Одиночки». Иначе рано или поздно наступит момент, когда примененные паттерны будут мешать применению новых.

Используйте кодовую базу проекта, где есть только «Одиночка».

В уроке 7 нам потребуется исходный код проекта без паттерна «Одиночка», поэтому сохраните и его.

Домашнее задание

Ссылка на проект SBomber

1. Применение паттерна Singleton

Задача: применить паттерн «Одиночка» для логирования событий в проекте SBomber (рефакторинг модуля MyTools.cpp).

В единице трансляции MyTools.cpp есть вторая группа функций (помимо функций для работы с экраном) для логирования событий в проекте SBomber. Рассмотрим их прототипы:

```
void __fastcall OpenLogFile(const std::string& FN);
void CloseLogFile();
void __fastcall WriteToLog(const std::string& str);
void __fastcall WriteToLog(const std::string& str, int n);
void __fastcall WriteToLog(const std::string& str, double d);
```

Видим функции, которые открывают и закрывают файл для ведения лога, и 3 функции для записи данных в лог-файл.

Необходимо выполнить рефакторинг этих функций и поместить их внутрь «Одиночки» (FileLoggerSingletone) по аналогии с тем, как это было сделано выше для объекта-одиночки ScreenSingletone. Так мы сможем использовать эту функциональность в стиле ООП.

2. * Применение паттерна «Заместитель»

Задача: использовать паттерн «Заместитель» в проекте SBomber в алгоритмах логирования.

Используйте структурный паттерн «Заместитель» в проекте SBomber в качестве обёртки для процедуры логирования в файл через FileLoggerSingletone. Пусть в начало строки логирования обёртка добавляет номер логируемого события, начиная с 1. Дайте название LoggerSingleton — его тоже можно сделать «Одиночкой». Возможно, в будущем мы захотим управлять записью логов в файл, по сети или в БД. Тогда мы быстро усовершенствуем LoggerSingleton: добавим другие виды логирования и возможность переключения между ними. Остальной код, который пользуется этим классом, трогать не придётся.

Глоссарий

- **Шаблон проектирования** или паттерн (design pattern) — повторяемый архитектурный приём, который решает конкретную проблему проектирования.
- **Рефакторинг** (англ. refactoring) — переработка исходного кода с целью сделать его проще и понятнее. Работа программы после рефакторинга никак не меняется.
- **SOLID** (сокр. от англ. single responsibility, open–closed, Liskov substitution, interface segregation и dependency inversion) — мнемонический акроним для пяти основных принципов объектно-ориентированного программирования и проектирования.
- **DRY** (англ. don’t repeat yourself) — «не повторяйся». Принцип снижения повторяемости информации при разработке ПО. Любая информация в рамках информационной системы должна иметь единственное, непротиворечивое и авторитетное представление.
- **KISS** (англ. keep it simple) — «не усложняй». Принцип избегания лишней сложности систем при их проектировании.
- **YAGNI** (англ. you aren't gonna need it) — «вам это не понадобится». Принцип отказа от избыточной функциональности при проектировании ПО.
- **UML** (англ. Unified Modeling Language) — унифицированный язык моделирования. Позволяет графически описывать объектные модели для целей при разработки ПО моделирования бизнес-процессов, системного проектирования и отображения оргструктур.
- **Диаграмма классов** (англ. class diagram) — структурная UML-диаграмма, которая отражает структуру иерархии классов системы: их кооперации, атрибуты (поля), методы, интерфейсы и взаимосвязи. Широко применяется для целей прямого и обратного проектирования, документирования, визуализации.
- **Взаимосвязь** — тип логических отношений между сущностями на диаграммах классов и объектов.
- **Зависимость** (dependency) — слабая форма отношения использования, при которой изменение в спецификации одного объекта влечёт за собой

изменение другого, причём обратное не обязательно. Возникает, когда объект является, например, параметром или локальной переменной.

- **Ассоциация** показывает, что объекты одной сущности (класса) связаны с объектами другой сущности таким образом, что можно перемещаться между классами. Общий случай композиции и агрегации.
- **Агрегация** — разновидность ассоциации, которая отражает отношения целого и его частей. Как тип ассоциации, агрегация может быть именованной. Одно отношение агрегации включает максимум 2 класса: контейнер и содержимое.
- **Композиция** (или «агрегация по значению») — строгий вариант агрегации. Подразумевает, что если контейнер будет уничтожен, то всё его содержимое будет также уничтожено. Графически выглядит как агрегация с закрашенным ромбом.
- **Обобщение** (Generalization) показывает, что один из двух связанных классов (подтип) является частной формой другого (надтипа), который называется обобщением первого. Сумчатые — это млекопитающие, млекопитающие — это животные.
- **Реализация** — отношение между элементами модели, при котором один элемент-клиент реализует поведение, заданное элементом-поставщиком.
- **«Одиночка»** (англ. Singleton) — порождающий шаблон проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет глобальную точку доступа к этому экземпляру.
- **«Заместитель»** (англ. Proxy) — структурный шаблон проектирования, который предоставляет объект для контроля доступа к другому объекту и при этом перехватывает все вызовы, то есть выступает контейнером.

Дополнительные материалы

Будут полезны при знакомстве с паттернами проектирования ПО:

- Шпаргалка по шаблонам проектирования
- Шаблон проектирования в Wiki

- Книга “Приемы объектно-ориентированного проектирования. Паттерны проектирования”
- Паттерны проектирования на CppReference
- Паттернах проектирования на RefactoringGuru
- Паттерны проектирования на Metanit

Используемые источники

1. Шаблон проектирования — Википедия
2. Что такое Паттерн?
3. SOLID в объектно-ориентированном программировании | CODE BLOG | Программирование
4. SOLID (объектно-ориентированное программирование) — Википедия
5. Don't repeat yourself — Википедия
6. https://ru.wikipedia.org/wiki/KISS_(%D0%BF%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF
7. YAGNI — Википедия
8. Design Patterns — Википедия
9. UML — Википедия
10. Диаграмма классов — Википедия
11. Следующий урок 13) Лучшие инструменты UML
12. Одиночка (шаблон проектирования) — Википедия
13. https://refactoring.guru/ru/design-patterns/singleton
14. «Заместитель» (шаблон проектирования) — Википедия
15. «Заместитель» Proxy