# Лекция ПКШ2018\_Л3

# Создание и использование классов

# 1. Введение

Главное отличие C++ от классических процедурных языков воплощено в пользовательских типах данных, или классах. Класс представляет собой абстрагированное описание объекта предметной области задачи, то есть в описании используются только те характеристики объекта, которые необходимы для решения конкретной разновидности задач. Класс является новым типом данных, определяемым вами или кем-то другим. Созданный класс может использоваться кем угодно, причем для этого даже не нужно точно знать, как он работает и как вообще создаются классы, достаточно знать его интерфейс.

Классы, созданные другими программистами, обычно собираются в библиотеки. Особенно заметная роль принадлежит стандартной библиотеке С++. Библиотечные классы можно рассматривать как встроенные типы данных, доступные для использования в программах.

Для использования библиотеки нужно выполнить следующие действия:   
1. Включить в программу заголовочный файл библиотеки.   
2. Скомпоновать библиотеку с исполняемой программой.  
3. Использовать в программе функции и переменные библиотеки.

## 1.1. Развитие абстрактных представлений в языках программирования

Процедурные языки программирования (например, С) могли рассматриваться как абстракции структуры компьютера (операция – команда, имя переменной – адрес ОП), а не структуры решаемой задачи (объекты предметной области). Программисту приходилось сопоставлять машинную модель («пространство решения») и модель решаемой задачи («пространство задачи», в котором определяется поставленная задача).

В языках программирования наряду с моделированием компьютера также существует другое решение — моделирование решаемой задачи (использование классов в объектно-ориентированных языках программирования).

## 1.2. Принципы хорошего программирования

Следующие принципы программирования обобщают многолетний опыт программистов в разработке программ. Их использование поможет вам с годами стать хорошим программистом, то есть человеком, который разрабатывает хорошие программы. Хорошая программа – это программа, которая уменьшает время собственной разработки и содержит код, который проще в сопровождении и имеет меньше дефектов.

Включение классов в объектно-ориентированные языки программирования как раз и направлено на упрощение реализации принципов хорошего программирования при разработке больших программных систем.  
 Собственно, главная цель объектно-ориентированных языков заключается в скрытии технических подробностей, чтобы программист мог творить на более высоком уровне.

Перечислим основные из этих принципов.

*- DRY (Don’t repeat yourself) Не делайте одну и ту же работу дважды.*

Наверное, самый фундаментальный принцип в программировании - это избегать повторения. Многие программные конструкции существуют исключительно для этой цели (т.е. циклы, функции, классы и другие). Как только вы начнете повторять себя (например, длинное выражение, серия похожих выражений, похожие сущности) создавайте новые абстракции.

*- KISS (Keep it simple, stupid!) Не усложняй, тупица*

Упрощение (и избежание сложности) всегда должно быть ключевой целью. Простой код занимает меньше времени чтобы его написать, имеет меньше ошибок, и его легче изменить.

*- Принцип Открытости/закрытости (Open/Closed Principle)*

Программные Сущности (классы, модули, функции) должны быть открыты для расширения и закрыты для изменений. Другими словами, не следует писать классы, которые люди могут изменять, создавайте классы, которые люди могут расширять. Это означает, что новое поведение должно добавляться только добавлением новых сущностей, а не изменением старых.

*- Пишите код для сопровождающего, а не для себя (Write Code for the Maintainer)*

Практически любой код, который стоит писать, предстоит поддерживать в будущем вам или кому-то другому. В будущем вы, когда вернётесь к коду, обнаружите что большая его часть совершенно вам незнакома, так что старайтесь писать так, как будто вы пишете для другого.

*- Принцип единственной ответственности (Single Responsibility Principle)*

Компонент кода (т.е. класс или функция) должен выполнять единственную хорошо определённую задачу.

*- Скрытие деталей реализации (Hide Implementation Details)*

Скрытие деталей реализации позволяет изменять реализацию кода компонента с минимальным затрагиванием любых других модулей, которые используют этот компонент.

## 2. Основные парадигмы ООП

## 2.1 Скрытая реализация (инкапсуляция)

Для удобства всех программистов можно условно разделить на создателей классов (тех, кто определяет новые типы данных) и прикладных программистов («потребителей», использующих типы данных в своих приложениях).   
*Прикладные программисты* собирают библиотеки классов и используют их для ускорения разработки приложения.   
*Создатель класса* стремится построить класс, который предоставляет прикладному программисту доступ лишь к тому, что необходимо для его работы, и скрывает все остальное. Почему?   
Скрытые аспекты класса не могут использоваться прикладными программистами, поэтому создатель класса может свободно изменять их, не беспокоясь о том, как эти изменения отразятся на других. Скрытая часть класса обычно представляет чувствительные «внутренности» объекта, которые могут быть легко испорчены прикладным программистом по неосторожности или от недостатка информации, поэтому скрытие реализации уменьшает количество ошибок в программе.

Скрытие реализации — одна из важнейших концепций ООП, ее важность трудно переоценить. В любых отношениях важно установить границы, которые должны соблюдаться всеми участниками этих отношений.

Для определения уровней доступа в C++ используются три ключевых слова: public, private и protected. Их смысл весьма прямолинеен — эти спецификаторы доступа указывают, кто может работать с объявлениями членов класса, следующими за ними. Фактически он разделяет «зоны ответственности» создателя класса и прикладного программиста. При попытке обратиться извне к закрытому (private) члену класса происходит ошибка компиляции. Спецификатор protected аналогичен private с одним отличием: производные классы могут обращаться к защищенным (protected) членам класса, но доступ к закрытым (private) членам для них запрещен. О наследовании и производных классах мы поговорим чуть позже.

## 2.2 Повторное использование реализации (наследование и композиция)

Созданный и протестированный класс должен (в идеале) быть достаточно самостоятельным, чтобы его можно было использовать повторно в других программах. Возможность повторного использования кода относится к числу важнейших преимуществ объектно-ориентированного программирования.

*Композиция.* В простейшем случае вы просто напрямую создаете объект соответствующего класса, однако существует и другой способ — включить объект этого класса внутрь нового класса. Это называется «созданием вложенного объекта». Новый класс может содержать сколько угодно объектов произвольных типов в любой комбинации, необходимой для достижения желаемого поведения нового класса. Методика создания нового класса из существующих классов называется композицией.   
На UML-диаграмме композиция обозначается линией с ромбом.

*Наследование* играет особую роль в объектно-ориентированном программировании и у неопытных программистов может сложиться впечатление, что наследование требуется в любых ситуациях. Это часто приводит к появлению громоздких, чрезмерно усложненных архитектур. При создании новых классов сначала подумайте, нельзя ли воспользоваться более простым и гибким механизмом композиции. А когда у вас появится практический опыт, станет вполне очевидно, в каких случаях лучше подходит наследование.

## 2.3 Наследование и повторное использование интерфейса

Класс не ограничивается определением общих свойств и методов для объектов этого класса, он также участвует в системе отношений с другими классами. Два класса могут обладать общими характеристиками и поведением, но один из них может обладать дополнительными характеристиками или обрабатывать дополнительные сообщения (или обрабатывать их иным образом). В механизме наследования сходство между классами выражается концепциями базовых и производных классов. Базовый тип имеет общие характеристики и аспекты поведения всех типов, производных от него. Программист создает базовый тип для представления важнейших особенностей объектов системы. Затем от базового типа наследуются производные типы, выражающие разные варианты реализации этих важнейших особенностей.  
Путем наследования программист строит иерархию типов, которая должна выражать решаемую задачу в контексте этих типов. На UML-диаграмме стрелка, показывающая отношение наследования, направлена от производного класса к базовому.   
*Формулировка решения в контексте задачи приносит огромную пользу. При использовании объектов диаграмма классов становится основной моделью, что позволяет перейти от описания системы в реальном мире к описанию системы на уровне программного кода. В этом примере проявляется простота объектно-ориентированного проектирования*.

Объявляя новый тип производным от существующего типа, вы создаете новый тип. Этот новый тип содержит все члены существующего типа, хотя члены со спецификатором private базового класса скрыты и недоступны для прямого доступа из методов производного класса, но они хранятся в памяти каждого объекта производного класса. Но что еще важнее — производный класс также воспроизводит интерфейс базового класса. Иначе говоря, все сообщения, которые принимались объектами базового класса, также будут приниматься объектами производного класса. Поскольку мы знаем, что класс определяется теми сообщениями, которые он принимает, можно сказать, что производный класс является частным случаем базового класса. *Эквивалентность типов* на базе наследования — одно из важнейших положений, необходимых для понимания сути объектно-ориентированного программирования.

Если просто объявить класс производным и не делать ничего более, методы интерфейса базового класса перейдут в производный класс. В этом случае объекты производного класса не только являются частным случаем базового класса, но и ведут себя точно так же, что не особенно интересно. *Существуют два способа модификации производных классов* по отношению к базовым.

Первый способ тривиален: в производный класс просто добавляются новые функции, не входящие в интерфейс базового класса. Если базовый класс не обладает возможностями, необходимыми для некоторого частного случая, он расширяется в производном классе. Такое простое и даже примитивное применение наследования часто оказывается идеальным решением проблемы, но при расширении следует хорошо подумать: «Нельзя ли ограничиться включением этих дополнительных функции в базовый класс?»

Второй, более важный, способ модификации производных классов основан на *переопределении методов* базового класса, то есть изменении поведения существующих функций базового класса.

Чтобы переопределить функцию, достаточно создать в производном классе новое определение функции. В сущности, это означает: «Мы используем ту же интерфейсную функцию, но в производном типе она будет делать нечто иное».

## 2.4 Точное и приблизительное подобие

В отношении наследования возникает один важный вопрос: должно ли наследование ограничиваться переопределением функций базового класса (без добавления новых функций, отсутствующих в базовом классе)? В этом случае производный тип в точности эквивалентен базовому классу, поскольку он обладает точно таким же интерфейсом. Следовательно, объекты производного класса могут свободно использоваться вместо объектов базового класса. Иногда это называется чистой подстановкой, или отношением *точного подобия*.

Иногда в производные типы приходится добавлять новые элементы интерфейса, что приводит к расширению интерфейса и созданию нового типа. Новый тип по-прежнему может использоваться вместо базового, однако эта подстановка не идеальна, потому что новые функции остаются недоступными для базового типа. Такая ситуация описывается как отношение *приблизительного подобия*; новый тип поддерживает интерфейс старого типа, но он также содержит другие функции, поэтому говорить о точном совпадении интерфейсов не приходится.

## 2.5 Взаимозаменяемость объектов и полиморфизм

При использовании иерархий типов часто требуется связывать объект не с фактическим, а с базовым типом. Это позволяет программировать независимо от конкретных типов. Так, в примере с макетом СУБД в классе DBTableSet функции работают с базовым классом DBTable независимо от того, связаны они с текстовыми или бинарными файлами.  
Они не заботятся о том, как объект обработает это сообщение. Такой код не зависит от введения новых типов — самого распространенного способа расширения объектно-ориентированных программ для обработки новых ситуаций.   
Возможность простого расширения программ посредством объявления новых производных типов играет важную роль, поскольку она существенно улучшает архитектуру программы с одновременным сокращением затрат на ее сопровождение. Тем не менее интерпретация объектов производных типов в контексте базового типа (DBTableTxt как DBTable) не обходится без проблем. Если функция должна приказать обобщенной таблице напечатать данные, на стадии компиляции еще неизвестно, какой фрагмент кода должен при этом выполняться (DBTableTxt::PrintDB() или DBTableBin::PrintDB()). Собственно, в этом и заключается вся соль — при отправке сообщения программист не желает знать, какой фрагмент кода будет выполняться. Функция вывода может применяться к обеим таблицам, а объект должен выполнить правильный код в зависимости от фактического типа. Итак, если компилятор не может заранее узнать, какой фрагмент кода будет выполняться, что же ему делать?

Ответ на этот вопрос кроется в одном из трюков объектно-ориентированного программирования. Компилятор не может сгенерировать вызов функции в традиционном смысле. В вызовах функций, сгенерированных обычным компилятором, применяется так называемое раннее связывание. Термин означает, что компилятор генерирует вызов функции с конкретным именем, а компоновщик преобразует этот вызов в абсолютный адрес выполняемой функции.   
Для решения этой проблемы в объектно-ориентированных языках применяется позднее, или динамическое, связывание. При отправке сообщения объекту функция, которой должно быть передано управление, не определяется до времени выполнения программы. Для выполнения позднего связывания компилятор C++ заменяет абсолютный вызов функции специальным фрагментом кода, который вычисляет адрес тела функции по информации, хранящейся в объекте. Таким образом, разные объекты по-разному ведут себя в зависимости от этого фрагмента. При получении сообщения объект сам решает, как его обработать. Виртуальные функции (то есть функции, наделенные гибкими возможностями позднего связывания) объявляются с ключевым словом virtual. Чтобы использовать виртуальные функции в программах, не обязательно понимать механику их вызова, но без виртуальных функций объектно-ориентированное программирование на C++ невозможно. В C++ необходимо помнить о добавлении ключевого слова virtual, потому что по умолчанию функции классов не используют динамическое связывание. В виртуальных функциях проявляются различия в поведении классов, принадлежащих к одному семейству. Эти различия заложены в основу полиморфизма. Вернемся к примеру с таблицами. Для демонстрации полиморфизма мы рассмотрим фрагмент кода, который будет игнорировать особенности конкретных типов и взаимодействовать только с базовым классом. Такой код изолируется от информации конкретных типов, он проще в программировании и сопровождении. А если в иерархию будет добавлен новый производный тип, то этот фрагмент будет работать с новой разновидностью типа DBTable так же, как он работал с существующими типами. Тем самым обеспечивается расширяемость программы.

Интерпретация объекта *производного* типа так, как если бы он относился к *базовому* типу, называется *повышающим приведением* типа. Термин «повышающее» происходит от типичной структуры диаграмм наследования, при которой базовый тип расположен сверху, а производные классы выстроены внизу. Таким образом, приведение к базовому типу соответствует перемещению вверх на диаграмме наследования.

В объектно-ориентированных программах обычно в том или ином виде задействовано повышающее приведение типа, потому что оно избавляет программиста от необходимости знать конкретный тип объектов, с которыми он работает. В данном случае мы говорим: «Таблица, я знаю, что ты умеешь самостоятельно обрабатывать вызовы PrintDB() и ReadDBTable(); сделай это и позаботься обо всех деталях. И это выглядит совершенно удивительно, потому что, как упоминалось выше, при компиляции компилятор не обладает информацией о типах, с которыми он работает. Тем не менее благодаря полиморфизму все работает правильно. Обо всех деталях в программе на С++ позаботится компилятор.

Программист должен знать лишь то, что этот механизм работает, и уметь применять его при проектировании. Если функция класса объявлена виртуальной, то при получении сообщения объект автоматически выполнит правильные действия, даже если при этом потребуется повышающее приведение типа.

## 2.6 Создание и уничтожение объектов

С технической точки зрения основными проявлениями ООП считаются абстрактные типы данных, наследование и полиморфизм, но существуют и другие, не менее важные аспекты. В этом разделе приводится их краткий обзор. Особого внимания заслуживают операции создания и уничтожения объектов. Где хранятся данные объекта и как определяется продолжительность его жизненного цикла? В разных языках программирования используются разные решения. В подходе, принятом в C++, важнейшим фактором считается контроль над эффективностью, поэтому программисту предоставляется выбор. Если он стремится добиться максимальной скорости выполнения, то место хранения и продолжительность жизненного цикла объектов определяются на стадии написания программы; для этого объекты размещаются в стеке или в статической памяти. Стеком называется область памяти, которая напрямую используется процессором во время выполнения программы. Переменные, хранящиеся в стеке, также называются автоматическими переменными. Статической памятью называется фиксированный блок памяти, выделяемый перед началом выполнения программы. При хранении данных в стеке или статической памяти на первое место ставится скорость выделения и освобождения памяти, очень существенная в некоторых ситуациях. Однако за нее приходится расплачиваться гибкостью, так как программист должен точно знать количество, жизненный цикл и тип объектов во время написания программы. Если задача плохо поддается прогнозированию (системы управления складом, системы автоматизированного проектирования и т. д.), это приводит к чрезмерным ограничениям. Во втором подходе объекты создаются динамически в специальном пуле памяти, называемом кучей. В этом случае программист до момента выполнения программы не знает, сколько объектов потребуется, каким будет их жизненный цикл или фактический тип. Эти решения принимаются «на месте» во время выполнения программы. Если программисту понадобится новый объект, он просто создает его в куче при помощи ключевого слова new. Когда объект становится ненужным, он ликвидируется при помощи ключевого слова delete. Поскольку операции с кучей производятся динамически на стадии выполнения, выделение памяти занимает гораздо больше времени, чем при создании объектов в стеке (выделение памяти в стеке часто сводится к одной команде процессора для смещения указателя стека). Динамический подход основан на разумном допущении, что для достаточно сложных объектов дополнительные затраты на поиск области памяти и ее освобождение не оказывают заметного влияния на общую продолжительность операции. Кроме того, более гибкий характер динамического выделения играет важную роль при решении общих задач программирования. Однако наряду с этими факторами существует и другой - *продолжительность жизни* объекта. Если объект создается в стеке или в статической памяти, продолжительность его существования определяет компилятор, автоматически уничтожая объект. Но если объект создается в куче, компилятор не располагает информацией о сроке его существования. В C++ программист должен самостоятельно уничтожить объект в своей программе при помощи ключевого слова delete. Впрочем, существует и другой вариант — в среде выполнения может поддерживаться механизм уборки мусора, который автоматически обнаруживает неиспользуемые объекты и уничтожает их. Конечно, уборка мусора существенно упрощает программирование, но, с другой стороны, она приводит к дополнительным затратам ресурсов. Механизм уборки мусора запускается периодически с интервалом в несколько секунд. Этот факт не соответствовал требованиям, принятым при проектировании C++, поэтому механизм уборки мусора не стал частью языка (хотя для C++ существуют уборщики мусора, разработанные сторонними производителями).

## 2.7 Обработка исключений

Со времен появления первых языков программирования обработка ошибок вызывала массу проблем. Разработать хорошую схему обработки ошибок чрезвычайно трудно, поэтому многие языки просто игнорировали эту проблему и перепоручали ее проектировщикам библиотек. В свою очередь, они предлагали компромиссные решения, основанные на обработке кодов возврата, которые работали, но во многих ситуациях легко обходились (как правило, их можно было просто игнорировать). Главным недостатком большинства схем обработки ошибок была их зависимость от того, насколько прилежно программистом выполнялись определенные правила, соблюдение которых не обеспечивалось на уровне языка. Если программист не проявлял должного усердия (как это часто бывает в спешке), все эти схемы становились бесполезными. Механизм обработки исключений переводит обработку ошибок на уровень языка программирования, а иногда даже на уровень операционной системы. Исключение представляет собой объект, который «генерируется» в месте ошибки и «перехватывается» соответствующим обработчиком исключения, предназначенным специально для обработки этого конкретного типа ошибки. Со стороны все выглядит так, словно при возникновении ошибки выполнение программы идет по другой, параллельной ветви. А это означает, что код обработки ошибок может быть отделен от кода нормального выполнения. Подобная изоляция упрощает программу, поскольку программисту не приходится постоянно проверять все возможные ошибки. Кроме того, сгенерированное исключение принципиально отличается от возвращаемого численного кода или флага, устанавливаемого функцией для обозначения ошибки, — эти признаки могут просто игнорироваться. Исключения не игнорируются, поэтому рано или поздно они заведомо будут обработаны. Наконец, исключения обеспечивают восстановление после аварийных ситуаций. Вместо завершения программы нередко удается исправить ошибку и продолжить работу, что значительно повышает устойчивость системы.  
В объектно-ориентированных языках исключения обычно представляются объектами класса «Исключение».

# 2. Разработка объектно-ориентированных программ

[*Никлаус Вирт*](https://ru.wikipedia.org/wiki/%D0%9D%D0%B8%D0%BA%D0%BB%D0%B0%D1%83%D1%81_%D0%92%D0%B8%D1%80%D1%82) (1934 г. Швейцария, автор языка программирования *Паскаль*) считает, что ООП — не более чем тривиальная надстройка над структурным программированием, и преувеличение её значимости, выражающееся, в том числе, во включении в языки программирования всё новых модных «объектно-ориентированных» средств, вредит качеству разрабатываемого программного обеспечения.

Не отрицая действительно имеющихся дополнительных накладных расходов на организацию работы ООП-программ, нужно, однако, отметить, что значение снижения производительности часто преувеличивается критиками. В современных условиях, когда технические возможности компьютеров чрезвычайно велики и постоянно растут, для большинства прикладных программ (в том числе для интерактивных информационных систем), вычислительная эффективность оказывается менее существенна, чем функциональность, скорость разработки и сопровождаемость. Лишь для некоторого, очень ограниченного класса программ (ПО встроенных систем управления, драйверы устройств, низкоуровневая часть системного ПО, научное ПО) производительность остаётся критическим фактором.

***Прежде чем начать программирование, необходимо выполнить проектирование программы, то есть дать ответы всего на три вопроса:***  
• Какие функции должна выполнять система?   
• Какие объекты будут использованы при реализации? (Как разбивать проект на составляющие)   
• Какими интерфейсами они будут обладать? (Какие сообщения должны отправляться каждому объекту).

Даже если у вас не будет ничего, кроме представлений об объектах и их интерфейсах, можно переходить к написанию программы. По различным причинам вам могут потребоваться другие описания и документы, но меньшим обойтись невозможно.

*Процесс разработки объектно-ориентированных программ делится на пять фаз* с дополнительной нулевой фазой, на которой просто принимается решение об использовании той или иной структуры процесса.

***Фаза 0 — составление плана***

Сначала следует решить, из каких этапов будет состоять процесс. На первый взгляд звучит просто (собственно, все формулировки такого рода будут звучать просто), но нередко подобные решения принимаются уже после начала программирования.  
 Если вы руководствуетесь планом «Поскорее взяться за программирование, а дальше ориентироваться по обстановке» — что ж, замечательно (при решении очевидных проблем иногда достаточно и этого). По крайней мере, осознайте, что ваш план звучит именно так.   
В результате планирования проект делится на части, с которыми удобнее работать и которые выглядят не столь устрашающе. Наличие промежуточных контрольных точек помогает сосредоточиться и направляет усилия программиста к этим контрольным точкам, не позволяя ему зациклиться на одной цели — «завершении проекта».   
Следовательно, даже если ваш план сводится к тому, чтобы поскорее взяться за программирование, все равно стоит пройти следующие фазы, задать и ответить на некоторые вопросы.

***Фаза 1* - *формулировка задачи (что делать).***

Функциональные требования определяют ключевые особенности системы и дают представление о некоторых основных классах, которые будут использоваться в вашей программе. В сущности, набор функциональных требований состоит из содержательных ответов на следующие вопросы:   
• Кто будет использовать эту систему?   
• Какие операции смогут выполнять операторы?   
• Как оператор будет выполнять ту или иную операцию?   
• Какие еще возможны варианты, если та же операция будет выполняться кем-то другим или у того же оператора появится другая цель (анализ вариантов)?   
• Какие проблемы могут возникнуть при выполнении операции с системой (анализ исключений)?

Например, если вы проектируете ИС «Библиотека», то функциональные требования будут описывать действия ИС в любой возможной ситуации. Каждая из таких ситуаций называется сценарием. Сценарий можно рассматривать как вопрос, начинающийся со слов: «Как поступит система, если...?»   
Например:   
- Как поступит система, если студент заказывает книгу, а в библиотеке нет свободных экземпляров этой книги?  
- Как поступит система, если студент заказывает книгу, а у него на руках есть книга, срок сдачи которой истек?

Диаграммы функциональных требований должны быть как можно проще, чтобы вы раньше времени не увязли в подробностях реализации.   
Можно ограничиться списком с несложными *UML*-диаграммами прецедентов, одна из которых приведена на рисунке 3.1.   
Фигурки на этом рисунке изображают «операторов» — обычно это люди или какие-либо независимые агенты (например, другая компьютерная система). Прямоугольник изображает границы системы, а овалы — описания отдельных операций, выполняемых системой. Линии, соединяющие операторов с операциями, изображают взаимодействия. В данном случае важна не конкретная реализация системы, а ее структура с точки зрения пользователя, поэтому даже для сложной системы функциональные требования не обязаны быть излишне сложными.

Эта фаза завершается разработкой документа «Техническое задание», содержание отдельных пунктов которого может быть в дальнейшем скорректировано по взаимному согласию заказчика и разработчика.

*Фаза 2 — как делать*

В этой фазе создается архитектура, описывающая классы и их взаимодействия.  
Признаком завершения фазы 2 является законченное описание объектов и их интерфейсов... или по крайней мере их большей части — некоторые из них остаются неучтенными и выявляются только во время фазы 3. Это вполне нормальное явление. Важно лишь, чтобы все объекты были рано или поздно выявлены. Хорошо, когда они обнаруживаются на ранней стадии процесса, но ООП обладает достаточной структурной устойчивостью, так что более позднее обнаружение не принесет особого вреда.   
Результат представляется в виде *UML*-диаграммы взаимодействия классов.

Возможно, вам также потребуется описать структуры данных в тех системах или подсистемах, в которых доминирующая роль отводится данным (например, при работе с базами данных).

*Фаза 3 — построение ядра (программирование системы классов)*

В этой фазе первоначальный набросок архитектуры преобразуется в компилируемый исполняемый код, который можно протестировать. Но самое главное, что прототип должен либо подтвердить, либо опровергнуть исходную архитектуру.

Фактически вы строите каркас, который будет наращиваться при последующих итерациях. Также при этом выполняется первая попытка системной интеграции и первое тестирование, а заказчик получает представление о том, как будет выглядеть его система и как идет работа. Одной из составляющих разработки системы является сравнение промежуточного результата с анализом требований и системными спецификациями (в той форме, в которой они существуют). Убедитесь в том, что результаты тестирования соответствуют всем требованиям и сценариям. Когда ядро системы заработает стабильно, можно двигаться дальше и добавлять новые возможности.

*Шаг 4 — итеративный перебор сценариев*

Получив работоспособное ядро, вы начинаете добавлять к нему новые возможности.   
Каждое функциональное требование представляет собой совокупность связанных функциональных возможностей, которые встраиваются в систему как единое целое.  
Итеративность позволяет обнаружить и решить критически важные проблемы на более ранней стадии; у заказчика появляется возможность откорректировать начальную постановку задачи; работа программиста становится более творческой, а сам проект лучше управляется.

*Шаг 5 — эволюция*

Эта стадия цикла разработки традиционно называлась «сопровождением».   
Эволюция обеспечивает тот промежуток времени, когда программа превращается из хорошей в замечательную, а проблемы, которые оставались непонятными при первой попытке, внезапно проясняются. Тогда же классы могут превратиться из разовых разработок в ресурсы многократного использования.

Вы должны быть уверены в том, что структура программы переживет все неизбежные изменения во время ее жизненного цикла и что эти изменения можно будет вносить быстро и просто.



Рисунок 3.1. UML-диаграмма прецедента «Выдача книги студенту»

Рисунок 3.2 UML-диаграмма базовой версии класса DBTableTxt

Рисунок 9.1 UML-диаграмма взаимосвязанных классов макета СУБД



Рисунок 3.4 UML-диаграмма класса DBTableBin