Основы модульности

Данная статья показывает подход к декомпозиции программного кода в языке программирования C++ при решении типовой задачи. Рассмотрено типовое «быстрое» или «студенческое» решение и показаны пути правильной объектно-ориентированной декомпозиции.

Дальнейшее рассмотрение проведём на примере разработки программы, выполняющей замеры скорости работы алгоритмов. В ходе выполнения основной задачи потребуется разработать дополнительный инструментарий – в частности, средство вывода графиков.
Безусловно, правильным решением было бы использование готовых компонент – в частности, средства построения графиков и диаграмм, но данная статья рассматривает не оптимальный способ написания программной системы, а методы модульной разработки.

Сразу оговорюсь, что данная статья описывает подход к решению задачи, а не полное её решение. Более подробно принятые сознательно ограничения указаны в конце статьи.

Постановка задачи

Требуется написать тест, оценивающий производительность ряда алгоритмов при изменении объёма исходных данных. Примером может служить определение времени работы алгоритмов сортировки.
Результаты измерений необходимо вывести в виде графика, чтобы можно было визуально определить вид зависимостей (время выполнения операции при заданном объёме данных) и определить относительную эффективность алгоритмов. При выводе графика необходимо обеспечить минимальные дополнительные элементы: оси координат, сетку координат, значения величин вдоль осей.

Замечания о среде разработки

В качестве среды разработки используется операционная система Windows, компилятор Visual Studio .NET 2005 и библиотека MFC. Однако принципиального значения для содержания статьи и понимания примеров они не имеют. Достаточно базового понимания используемых в статье понятий, чтобы разобраться в том минимальном программном интерфейсе указанных систем, который используется в статье. Поэтому указанную здесь специфику можно игнорировать.

Шаг 0. Постановка мышления

Программирование является инженерной дисциплиной. Из этого следует несколько важных выводов:

1. Чаще всего при разработке программы приходится решать инженерные задачи. То есть придумать или изобрести решение. Пусть оно даже много раз было изобретено до вас. Реализуя программу и не имея готовых решений, вы выполняете творческую работу исследователя. Такая работа подразумевает проведение экспериментов, построение прототипов, изменение видения системы в ходе её появления.
2. Разрабатывая модули или подсистемы, проектировщик должен думать как шахматист. Основным его вопросом должно быть «А что, если?». То есть каждое решение должно анализироваться на предмет возможного изменения внешних условий. Чем глубже будет проведён анализ, тем меньше вероятность допустить ошибку.

3. Ошибки (постановки задачи, проектирования, кодирования) должны исправляться сразу при их обнаружении, поскольку чаще всего исправление ошибок на более поздних стадиях обходится значительно дороже.

Любая программа по определению является сложной системой. И тут не важно, сколько в ней строк – сто или сто тысяч. У каждого свой уровень сложности. Для профессионального программиста сложная система начинается с десятков тысяч строк. Для начинающего – с сотен строк. Под сложностью можно понимать много аспектов. В рамках данной статьи основные критерии сложности программной системы будут таковы:

1. Программу сложно рассматривать как единое целое на одном уровне понимания.
2. Программа не поддаётся анализу или доказательству правильности работы существующими математическими методами.

Признав, что программа является сложной системой, мы получаем в пользование типовой инструментарий работы со сложными системами. Основные тезисы его таковы (по книге Гради Буча «Объектно-ориентированный анализ и проектирование»):

1. Сложные системы являются иерархическими и состоят из взаимозависимых подсистем, которые в свою очередь также могут быть разделены на подсистемы, и т.д. вплоть до самого низкого уровня.
2. Выбор, какие компоненты в данной системе являются элементарными, относительно произволен и в большей степени оставляется на усмотрение исследователя.
3. Внутрикомпонентная связь обычно сильнее, чем связь между компонентами. Это обстоятельство позволяет отделять «высокочастотные» взаимодействия внутри компонентов от «низкочастотной» динамики взаимодействия между компонентами.
4. Иерархические системы обычно состоят из немногих типов подсистем, по-разному скомбинированных и организованных.
5. Любая сложная система является результатом развития работавшей более простой системы. Сложная система, спроектированная «с нуля» никогда не заработает. Следует начинать с работающей простой системы.

Шаг 1. «Студенческое» решение

Типовое решение, которое мне приходилось видеть – код, написанный методом «грубой силы». В большей степени напоминает песню акына – «что вижу, о том пою» с одним отличием. Тут больше подходит «о чём вспомнил, о том пою».

Я не буду приводить реальный студенческий код со всеми его ошибками и недочётами. Я лишь опишу его основные отличительные черты.

Автор запускает Visual Studio, создаёт при помощи «мастера» MFC-проект и вспоминает, что рисование происходит в методе OnDraw класса, унаследованного приложением от CView. Здесь начинается исследование того, как в MFC/Windows рисовать на экране.

Это правильный этап. Исследование. Это необходимо сделать, чтобы понять программный интерфейс (API) системы, с которой придётся работать. Вот только превращать исследовательский стенд в готовый продукт не надо. Автор делает это, постепенно, шаг за шагом наращивая функционал.

Типовые проблемы кода:

1. В коде смешаны все части функционала: расчёты, вывод на экран, обработка пользовательского ввода. Код представляет собой монолит, который можно рассматривать только целиком. Разделить его на части можно лишь условно, либо выполнив определённый объём работ.
2. Код множественно продублирован. Одни и те же фрагменты повторяются по нескольку раз.
3. Идентификаторы (имена объектов, переменных, функций) назначены бессистемно. Можно встретить что угодно от «ааа» до “peremennaya”.

4. Возможные ошибочные ситуации (деление на ноль, извлечение корня из отрицательной величины, аргументы и возвращаемые значения) не проверяются.
5. Освобождения использованных ресурсов не производится. Чаще всего программа вообще годится только для однократного выполнения задачи – загрузить новые данные или повторить эксперимент, не перезапустив программу, чаще всего невозможно.

Шаг 2. Модульный подход и проектирование

Начнём с разделения системы на модули или объекты. Первый компонент, который очевиден – график функции. Логично отделить его от других задач (получения данных эксперимента и т.д.) и сделать таким, чтобы его можно было впоследствии использовать при решении других задач. Я буду показывать не результаты проектирования и кодирования, а сам процесс. То есть статья будет показывать, как ведётся итеративная разработка, какие вопросы необходимо задавать и как на них можно отвечать.

Первое требование к исходному коду – понятность. Заметьте, не работоспособность или правильность, а понятность. Причина проста – программный код пишется не для компьютеров, а для людей. С кодом придётся работать людям. Как минимум – автору. Но зачастую – другим программистам. И код им должен быть понятен. Легко понятен.

Первое что требуется сделать – создать в проекте новый заголовочный файл. Назовём его Graphic.h. Начать необходимо не с кода, а с комментария о назначении файла, его авторе и т.д. Это не только полезная справочная информация. Это ещё дополнительный контроль содержимого заголовочного файла. Если в комментарии правильно написано, какой цели служит модуль и каков его состав, то значительно уменьшается вероятность попадания в модуль лишних сущностей.

Далее – директива #pragma once или любой другой любимый вами страж включения – конструкция, необходимая, чтобы заголовочный файл не компилировался дважды при многократном включении в файл исходного текста.

Теперь непосредственно код первой итерации. Запишем то, что мы знаем о необходимом нам классе графика на C++. Вначале напишем интерфейсы, которые должны быть предоставлены пользователю.

```
class Graphic {
public:
  Graphic();

  // MANIPULATORS
  void ClearContent();
  void AddValue(double x, double y);

  void ShowGrid(bool is_on);
  void AlignAxisToBottomLeft(bool is_on);

  void Draw(CDC *,const CRect& extent) const;
};
```

В конструкторе будет происходить инициализация внутренних данных графика. Метод ClearContent необходим для того, чтобы иметь возможность удалить все накопленные данные. Метод AddValue позволяет добавлять элементы данных.

Имена методов (как и функций) в основном начинаются с глаголов, определяющих действие. То есть «ОчиститьСодержимое» или «ДобавитьЗначение».

Архитектурный вопрос

Правильно ли архитектурное решение, при котором график будет владеть данными, которые он отображает? Можно ли реализовать интерфейс таким образом, чтобы данные передавались непосредственно при рисовании (метод Draw)?

```
void Draw(CDC *, const value *data, size_t count, const CRect& extent);
```

На первый взгляд это – хорошая идея. Исчезает высокочастотный метод AddValue, который придётся вызывать многократно. Исчезает необходимость копирования данных. Но у этого решения есть и недостатки:

1. Не факт, что клиент будет иметь данные в подготовленном виде на момент вызова Draw. Вполне возможно, что значения будут появляться по мере работы программы. В этом случае буферизировать данные придётся на стороне клиента. То есть работу всё равно придётся выполнить, но уже у многих пользователей, а не в одном классе системы.
2. Если клиент и подготовит данные, то передавать их копию не имеет смысла – методу Draw должен будет передаваться константный указатель на контейнер (например, массив) с элементами. А это сразу же создаёт множество проблем с владением и временем жизни: контейнер должен существовать и не изменяться всё время работы метода Draw. А по завершении его работы контейнер должен быть освобождён. Это добавляет проблем пользователю класса Graphic.
3. При рисовании графика сортировки элементов не избежать, поэтому выигрыша оттого, что элементы, казалось бы, не копируются, мы не получим. Копию придётся сделать.

Вывод: Решение с передачей данных при рисовании не является ошибочным (в других случаях оно может быть использовано), но в данной задаче неэффективно.

Метод *ShowGrid* определяет будет ли рисоваться координатная сетка.
Метод *AlignAxisToBottomLeft* указывает, будут ли координатные оси насильно располагаться в левом нижнем углу графика или они будут проходить через начало координат.

Метод Draw служит для отображения графика. Первый аргумент – контекст устройства, на которое будет выводиться изображение. Для тех, кто не знаком с MFC скажу, что это тот самый объект, который предоставляет API для рисования на экране. Большего в рамках данной статьи знать не требуется. Второй аргумент – область, в которой необходимо рисовать график. Она нужна по двум причинам: первая - контекст устройства сам не знает размеров клиентской области окна, вторая – пользователь может потребовать выводить график в какой-то части окна, а не во всём окне.

Архитектурный вопрос

Можно ли сделать метод SetExtent, который позволит единожды установить область рисования и не передавать этот параметр каждый раз при рисовании?

На первый взгляд это – хорошая идея. Явно, что данный метод будет вызываться намного реже, чем метод Draw. Это позволит оптимизировать число передаваемых данных.

Но в реальности это – плохое решение. И вот почему. Дело в том, что информация

об области рисования необходима к моменту начала рисования. То есть должны быть гарантированы два предусловия выполнения Draw: область рисования должна быть установлена и она должна быть актуальной.

То есть кто-то (клиент) должен гарантировать, что он вызовет SetExtent до первого вызова Draw, и что он будет вызывать его каждый раз, когда область рисования может измениться (например, при изменении размеров окна).

Фактически, разделив функцию Draw на две, мы получили нагрузку в виде проблемы синхронизации вызовов. Сомнительный выигрыш в числе передаваемых параметров обернётся проблемами намного более сложными. В дополнение замечу, что время работы метода Draw намного больше времени передачи дополнительного значения (и даже времени его запроса клиентом у Windows).

Метод Draw объявлен как константный потому, что хотя он и меняет картинку на экране, содержимое объекта (C++) не меняется. Любой метод, не меняющий содержание объекта должен объявляться как константный.

Пояснение
Передача области рисования (extent) по константной ссылке является общепринятым механизмом передачи больших структур данных в C++. Передаётся не копия объекта, а сам объект (его альтернативное имя, как говорится в определении ссылки в C++). Это позволяет избежать затрат процессорного времени и памяти на ненужное копирование объекта. Передача по ссылке аналогична передаче указателя за тем исключением, что невозможна передача нулевого указателя. То есть интерфейс подразумевает, что объект всегда должен быть. Константность гарантирует, что модуль не будет изменять полученные данные. Без объявления ссылки константной модуль клиенту пришлось бы гадать – не изменяет ли модуль размеры переданной ему области рисования.

Реализация

Поскольку мы определили два независимых режима рисования графика, нам необходимо обеспечить переменные для хранения соответствующих режимов в классе Graphic и тут всё просто:

```
class Graphic {
  bool  is_grid_shown;
  bool  is_axis_aligned_to_bottom_left;
```

Название переменных отражает их назначение. Префикс «is» показывает, что это двоичные переменные, означающие включен ли соответствующий режим. В отличие от имён методов после префикса указывается объект, свойство которого описывается, а затем – само свойство.

Немного сложнее обстоит со свойством, описывающим число линий сетки. Не желая усложнять интерфейс класса, принимаю два решения:

- Число линий сетки по обеим осям будет одинаково
- Число линий будет задаваться в классе и не будет доступно для изменения пользователю.

Первое допущение легко исправить в случае необходимости. Поговорим о втором. С одной стороны может показаться, что в профессионально выполненном классе, строящем графики, такая настройка должна быть доступна пользователю. Но на самом деле – это внутреннее

решение класса о том, как произвести оформление графика. Число линий может быть вычислено на основании размеров области рисования или другой информации. Поэтому нет никакой необходимости загромождать интерфейс класса такой функциональностью. Существует способ предоставить такую возможность пользователю, не изменяя интерфейс класса, но описание её выходит за рамки данной статьи. Заканчивая рассмотрение этой темы, подскажу вопрос, который вы должны задать себе при проектировании: «Используя построение графиков на компьютере – часто ли я настраивал сетку координат?»

Второй вопрос, относящийся к этому свойству – какое значение оно должно хранить и как должна называться переменная. Первое решение назвать переменную grid_lines и хранить в ней число линий сетки оказывается неудачным, потому, что при всех последующих расчётах нам потребуется не число линий, а число ячеек (интервалов) сетки. А их, как известно, на один меньше, чем линий. В результате в коде повсеместно появятся конструкции вида (grid_lines-1) и неочевидные проверки вида if (grid_lines < 1). Если вы изначально приняли неправильное решение с содержимым переменной такие «мелкие неприятности» должны стать для вас индикаторами того, что необходимо изменение в коде.
Правильно будет назвать переменную grid_cells и хранить в ней число ячеек (интервалов) сетки.

Представление данных для графика

Данные графика должны храниться в контейнере, обеспечивающем хранение однородных данных. Выбор правильного контейнера для хранения данных определяет простоту и эффективность алгоритмов, которые будут использовать эти данные при решении целевой задачи.

Рассмотрим начальные условия и выводы, которые из них следует сделать.

Объём данных заранее неизвестен
Заранее неизвестно, сколько значений будет представлено. К тому же они могут меняться даже в ходе работы с графиком. Исходя из этого, мы не можем выделить фиксированный объём памяти. Нам требуется контейнер, позволяющий динамически изменять размер при добавлении элементов.

Не предусмотрено удаление элементов
Не предусмотрена вставка элементов в затребованной позиции
Значения будут помещаться в контейнер, но не потребуется удалять элементы. Пользователь не располагает информацией о том, как хранятся элементы в контейнере и не может требовать вставки элементов в заранее заданные позиции контейнера. Данные утверждения дают основание для отказа от связанного списка, как от возможного контейнера для хранения данных. Он хорошо справляется с задачей добавления элементов и обеспечения работы с произвольным числом элементов, но ценой за это являются дополнительная память для хранения указателя на следующий элемент, дополнительное время при переходе между элементами.

При выводе графика потребуется получать упорядоченные по оси x значения
Отсюда следует важный вывод: нам потребуется упорядоченность значений, которую можно обеспечить одним из двух возможных способов: либо помещать значения в контейнер, соблюдая упорядоченность, либо упорядочить контейнер перед началом вывода данных. Третий вариант – выбирать в неубывающем порядке элементы из неупорядоченного контейнера – исключаем сразу ввиду неэффективности.

Архитектурный вопрос
Какой из вариантов выбрать – поддерживать упорядоченность при добавлении или сортировать при выводе? Это почти равные варианты и в таких случаях выбирать приходится исходя из следующих соображений:

- Эффективность в отношении производительности
- Эффективность в отношении расходуемой памяти
- Простота поддержки в коде

Приоритет каждого из этих требований зависит от конкретной задачи.
Посмотрим со стороны производительности. Тут всё будет зависеть от того, что планируется делать чаще – рисовать график или добавлять данные.

Если чаще будут добавляться данные, то логично отложить упорядочение на стадию рисования – тогда не придётся тратить время при добавлении каждого элемента. С другой стороны чаще всего рисовать надо «как можно быстрее», поскольку запрос на обновление экрана обычно приходит как раз в тот момент, когда надо уже рисовать и любая задержка будет видна пользователю.

Пока будем исходить из того, что нам требуется чаще добавлять данные, и скорость рисования графика не так важна. Если в вашем случае условия иные – вывод будет обратным.

С точки зрения расходуемой памяти самой оптимальной структурой будет массив (или вектор) элементов. Здесь всё очевидно. Минимум накладных расходов.

С точки зрения простоты поддержки решения в коде. Вариант с упорядочением перед рисованием потребует одного вызова функции std::sort. Вариант с упорядочиванием при добавлении элемента потребует либо более сложного кода, либо более дорогой структуры данных.

В дополнение, он чреват сюрпризами. К примеру, начинающий программист может выбрать решение на основе ассоциированного массива (std::map), как наиболее простое – контейнер map поддерживает автоматическое упорядочение элементов по ключам (значение x).

```
std::map<double,double> values;
```

Ошибка, привносимая этим решением, достаточно сложно обнаружима. Дело в том, что этот контейнер хранит только уникальные ключи. То есть он не может хранить две точки с одной координатой x. При помещении нового значения с имеющимся уже значением с координатой x, старое будет утеряно.

Безусловно, это не проблема контейнера или архитектурного решения на данном уровне. В данном случае вообще следует понять, как рисовать график, если имеются две точки с одинаковым значением по оси x. Этот вопрос надо было поднимать раньше – когда было принято решение, что пользователь может передавать точки графика в произвольном порядке. Тогда надо было решить, что делать в такой ситуации. Но сейчас ситуация не упрощается – используя std::map мы получаем трудноуловимую ошибку, которая будет иметь достаточно сложное воспроизведение.

В этом случае исправит ситуацию решение использовать std::multimap, который для одного ключа позволяет хранить несколько значений. Но код при этом уже перестанет

быть простым и понятным.

Исходя из изложенного выше, имеет смысл остановить выбор на структуре std::vector, которая обеспечивает хранение данных в виде вектора. Отличие данной структуры от классического массива в том, что она обеспечивает возможность динамического увеличения размера при добавлении данных.

На данном этапе решение будет выглядеть так:

```
class Graphic {
  struct Value {
    double x,y;
  };
  std::vector<Value> values;
```

Value является структурой, а не классом, поскольку нет никакой необходимости в сокрытии полей и обеспечении внутренней целостности. По сути Value не является объектом – она является набором полей данных, рассматриваемых совместно.

Определение структуры Value внутри класса Graphic обеспечивает локализацию области видимости Value – это имя не «засоряет» глобальное пространство имён. Такое решение также добавляет понятности имени Value – Graphic::Value читается более понятно – «ЗначениеГрафика». То есть задаётся контекст.

Код

Рассмотрим основной метод класса – Draw. Он обеспечивает рисование графика на экране. Сигнатура метода:

```
void Graphic::Draw(CDC *dc,const CRect& extent)
```

обязывает нас в первой строке проверить валидность переданного указателя:

```
assert(dc);
```

Безусловно, вероятность передачи нулевого указателя на контекст устройства в профессионально разработанном приложении не велика, но мы понимаем, что в клиентской части может быть «студенческий» код, в котором соответствующую проверку не выполнили на стадии получения контекста устройства. Поэтому о неприятностях лучше узнать заранее.

Следует помнить, что проверка утверждений (assert) производится только в отладочной сборке программы и, поэтому не будет выполнена в финальной (Release) версии программы. В случае, если вы считаете, что в проверенной и поставляемой пользователю версии приложения может быть передан нулевой указатель на контекст устройства – можно дополнить защиту оператором if с соответствующей проверкой. Дополнить, но не заменить assert, поскольку последний позволяет значительно упростить отладку в случае возникновения ошибочной ситуации.

Поскольку дальнейшие действия по рисованию графика не имеют смысла в случае, если в наборе данных менее одного элемента, данное условие следует проверить:

```
if (values.size() < 1) return;
```

С момента написания данной строки данное требование становится узаконенным предусловием для всех остальных методов, которые будет вызывать Draw. То есть теперь гарантировано, что число элементов не меньше одного. Дальше это можно не проверять.

Теперь требуется упорядочить элементы массива. Главное – не начинать сразу же писать свой алгоритм сортировки. Есть уже готовое решение и оно называется std::sort. Остаётся только «научить» стандартный алгоритм упорядочивать объекты типа Graphic::Value, поскольку стандартный алгоритм не знает, как узнать, какой из них меньше, а какой – больше. Для встроенных типов всё очевидно. В случае пользовательского типа необходимо предоставить информацию отдельно.

Типовое предлагаемое решение состоит в перегрузке operator < для класса Value:

```
  struct Value {
    double x,y;
  public:
    bool operator < (Value& const &other) const;
  };
```

Решение типовое и в большинстве случаев – правильное. Как минимум в том, что тип Value будет вести себя как встроенный тип в том смысле, что у него теперь определён operator < и можно будет сравнивать значения типа Value привычным нам способом.
Однако, оно не даёт чёткого представления о том, по какому критерию производится сравнение. В рамках данной задачи лучшим решением является создать статическую функцию

```
struct Value {
  double x,y;
public:
  static bool is_less_by_x(Graphic::Value const &left,
                           Graphic::Value const &right);
};
```

Имя функции теперь явно указывает, что объект Value признаётся меньшим, если значение его компоненты x будет меньше. Функцию придётся сделать статической и передавать ей два параметра, поскольку таковы требования к предикату функции std::sort.

Пояснение
Статический метод отличается от обычного тем, что не требует для вызова объекта класса. То есть он может быть вызван как Value::is_less_by_x(value1,value2). По сути он является методом только в том смысле, что относится к указанной области видимости и получает доступ к членам класса. В отличие от обычного метода он также не получает указателя this, указывающего на объект, для которого он был вызван. Поэтому для сравнения двух объектов типа Value в необходимо передать два объекта в качестве параметров, а не один, как это делается в operator <, где левый аргумент передаётся неявно как this.

Архитектурный вопрос
Можно ли было сделать этот метод не членом класса Value? Нет, потому что это привело бы нас к двум плохим альтернативам:

- Либо функцию пришлось бы сделать методом класса Graphic, что нелогично и не имеет никакого смысла.
- Либо функцию пришлось бы сделать глобальной. Но, поскольку Value определена в private-секции класса Graphic, доступа к объектам класса Value у неё бы не было.

Поэтому единственный разумный вариант – сделать её статическим методом класса Value.

Итак, предоставив функцию, позволяющую сравнить два элемента типа Value, мы можем приступить к сортировке:

```
std::sort(values.begin(),values.end(),Graphic::Value::is_less_by_x);
```

В данном случае вызывается стандартный алгоритм сортировки, который упорядочивает контейнер, используя заданную нами функцию для сравнения элементов. В терминологии STL (стандартной библиотеки шаблонов) такая функция называется *предикатом.* В STL *предикат* – это функция или объект (функтор), принимающая объект или объекты и возвращающая логическое значение как результат.

Архитектурный вопрос
Если предположить, что рисование графика в некоторых случаях будет производиться без обновления данных (данные уже загружены, а изображение надо перерисовывать в связи с вызовом запросом операционной системы) – возможно ли запомнить, что данные уже были упорядочены и не вызывать алгоритм std::sort, если данные не менялись?
Тут есть несколько вариантов. Во-первых, следует учитывать, что вызов std::sort для уже упорядоченной последовательности обходится значительно дешевле (в несколько раз). Вполне возможно, что в условиях вашей задачи этого аргумента будет достаточно.
Если всё же хочется большего быстродействия – можно перед вызовом сортировки проверить, упорядочена ли уже последовательность:

```
bool is_ascending=true;
for (size_t i=1; i < values.size(); ++i)
  if (Value::is_less_by_x(values[i],values[i-1])) {
    is_ascending=false;
    break;
  }

if (!is_ascending)
  std::sort(values.begin(),values.end(),Value::is_less_by_x);
```

Это решение наиболее универсальное – оно позволяет не только избежать вызова сортировки, если данные не менялись, но и избежать вызова, если пользователь всегда вводит данные в нужном порядке, что бывает чаще всего. Цена этого решения – проверка упорядоченности массива при каждом выводе на экран графика.

Если слишком велика и эта цена – можно завести флаг, указывающий, упорядочена ли последовательность и взводить его после сортировки. Относительно дешёвое с точки зрения быстродействия решение имеет оборотную сторону – при всяком изменении данных надо не забывать сбрасывать этот флаг. Это кажется несложным сейчас, когда данные меняются в одной точке – в методе AddValue. Но с развитием функционала класса ситуация может измениться и данные могут меняться и в другом месте. Будет легко забыть про флаг, что приведёт к трудно обнаружимым ошибкам.

Но эта тема напрямую не относится к вопросу архитектуры класса и больше затрагивает тему оптимизации, которой будет посвящена отдельная статья.

Далее необходимо сформировать области, в которых будут рисоваться оси, сетка, график и метки осей.

Архитектурный вопрос
Какая из подсистем (методов, функций) должна принимать решение о расположении компонент изображения на экране? Должен ли это делать каждый модуль, рисующий часть графика или это должно делаться в модуле, отвечающем за рисование всех компонент.
Правильный ответ – размеры областей должны вычисляться тем модулем, который вызывает конкретные рисующие модули, поскольку только он владеет картиной в целом. Только он знает, как расположить компоненты рисунка. Он играет роль дирижера. В более сложных вариантах компоновки возможна ситуация, когда дирижёр предлагает формат расположения дочерним методам, а они сообщают свои потребности в занимаемом пространстве в зависимости от размера шрифтов и других данных. В этом случае дирижёр собирает эту информацию и принимает решение о наиболее оптимальном расположении.

Определим размер полей вокруг координатной сетки, где будут располагаться метки:

```
const double margin_space=0.1;
```

Архитектурный вопрос
Правильно ли было располагать определение margin_space в этом методе?
Сомнения вызываются тем фактом, что margin_space может оказаться настройкой, управление которой хочется передать пользователю. В любом случае, даже если это невидимая клиенту величина, хотелось бы все подобные параметры держать в одном месте, чтобы их было удобнее настраивать, не производя поиски по всему коду. Как, например, это сделано в случае с grid_cells. Переменная объявлена членом класса и инициализируется в конструкторе.
На самом деле всё сделано правильно. Основное отличие margin_space от grid_cells состоит в том, что margin_space используется только в одном месте кода. То есть данная величина не требуется нигде более за рамками метода Draw. Следовательно, нет смысла её хранить как поле класса.
Дополнительно можно указать на то, что в реальном коде величина margin_space будет вычисляться исходя из других параметров, а не задаваться жёстко. Но обычно такие улучшения будут вноситься уже по ходу развития подсистемы.
Последний довод – margin_space определена максимально локально – точно перед точкой использования, что делает её определение максимально эффективным.

Далее хочется предостеречь от типовой ошибки. Чаще всего дальше я вижу следующий код:

```
int width_margin = (extent.right-extent.left)*margin_space;
int height_margin =(extent.bottom-extent.top)*margin_space;
```

В этом коде сразу две помарки:

- Он выдаёт предупреждения, поскольку значение типа double, которое получается в результате вычислений, присваивается переменной типа int. И компилятор предупреждает, что произойдёт потеря точности.
- Нет смысла писать так много кода, достаточно посмотреть в документацию по классу CRect. Там уже есть возможность запросить ширину и высоту.

Вот правильный код:

```
int width_margin = int(extent.Width()*margin_space);
```

```
int height_margin = int(extent.Height()*margin_space);
```

Поскольку область рисования (extent) передана по константной ссылке, мы не можем изменить её, чтобы передать функциям, рисующим компоненты графика. Придётся сделать локальную копию.

```
CRect active_area_extent(extent);
```

И снова не забываем про документацию к CRect. Вместо, казалось бы, очевидного и логичного

```
active_area_extent.left  += width_margin;
active_area_extent.right -= width_margin;
active_area_extent.top   += height_margin;
active_area_extent.bottom-= height_margin;
```

можно просто написать

```
active_area_extent.DeflateRect(width_margin,height_margin);
```

Согласитесь, что в данном случае разница уже более очевидна – одна строка вместо четырёх, не надо думать, что вычитать, а что прибавлять и намного меньше вероятность ошибиться и запутаться. К тому же при сопровождении такой код намного проще читать.

Далее начинаем непосредственно процесс рисования:

```
if (is_grid_shown) DrawGrid(dc,active_area_extent);
```

Если пользователь включил рисование сетки, то вызывается метод рисования координатной сетки, которому передаются контекст устройства и область, в которой должна быть нарисована сетка.

После этого следуют вызовы

```
DrawAxis(dc,active_area_extent);
DrawGraph(dc,active_area_extent);
```

обеспечивающие рисование координатных осей и непосредственно графика.

Оси координат

При описании структуры данных для представления осей координат начальный этап проектирования вполне мог выглядеть так:

```
class Graphic {
  double axe_x_lower_limit, axe_y_upper_limit,
         axe_y_lower_limit, axe_y_upper_limit;
```

на этом этапе повторение может не настораживать. Но после добавления

```
bool is_axe_x_auto_limits, is_axe_x_auto_limits;
```

(для хранения признаков автоматического определения минимального и максимального значения по осям) уже стоит задуматься о дублировании данных. Хорошо, что у нас пока только две оси.

Финальную точку в размышлениях должно поставить появление методов

```
void SetAxeXLimits(float low, float high);
void SetAxeYLimits(float low, float high);
void SetAxeXAutoLimits(bool is_on);
void SetAxeYAutoLimits(bool is_on);
```

Такое дублирование уже не только добавляет работы при программировании, но и усложняет интерфейс для читающего.

Поэтому должен быть сделан следующий вывод:

```
class Graphic {
public:
  struct Axe {
    std::string name;
    bool  is_auto_limits;
    double lower_limit, upper_limit;
  public:
    Axe();

    void SetName(const std::string& name);
    void SetLimits(float low, float high);
    void SetAutoLimits(bool is_on);
};
```

Почему тип «ось» сделан структурой, а не классом? Потому, что на данном этапе проектирования данная сущность не обладает собственным поведением. Единственная возможность для данных Axe оказаться в противоречивом состоянии – это нарушить условие upper_limit >= lower_limit. Проверять это условие можно в методе SetLimits, а вероятность того, что оно будет нарушено при прямом доступе к членам структуры не настолько велика, чтобы ради этого писать большое число методов доступа.

Теперь вопрос, о том, как дать доступ клиентам класса Graphic к осям. На стадии разработки класса я предлагаю выбрать наиболее простое решение – объявить их в public секции класса Graphic. Это противоречит принципам ООП и построения интерфейсов, однако значительно упрощает интерфейс и работу с классом. На данном этапе вполне можно остановиться на таком решении. Когда в дальнейшем потребуется более полная изоляция класса – вопрос решится сам собой и намного проще.

Следует понимать, что вынос осей в public-часть класса сделан после тщательного выбора между усложнением интерфейса (и увеличением объёма кода) и обеспечением безопасного интерфейса класса. В общем случае лучше скрывать члены данных класса, если на иное решение нет веских оснований.

Помещение осей в private часть класса привело бы к одной из двух проблем.
Либо было бы сделано малоэффективное:

```
private:
  Axe   axe_x,axe_y;
public:
  Axe&  GetXAxe();
  Axe&  GetYAxe();
```

которое почти не снимает проблем, возникающих при выносе осей в public часть класса, либо пришлось бы создать ряд функций, которые вызывали бы соответствующие функции структуры Axe (такие функции называются «гейтами» - от англ. *gates* – ворота):

```
void SetAxeXName(const std::string& name) {
  axe_x.SetName(name);
}

void SetAxeXLimits(float low, float high) {
  axe_y.SetLimits(low,high);
}
```

и т.д., что значительно ухудшило бы читаемость класса и, что ещё хуже – вынесло бы интерфейс «оси» на уровень интерфейса «графика», что захламило бы его до невозможности использования.

Шаг N-1. Насколько глубока кроличья нора?

Интерфейс класса сформирован. Показана и реализация метода верхнего уровня *Draw*. С точки зрения реализации дальнейшее повествование выходит за рамки данной статьи, хотя оно также очень интересно и заслуживает отдельного внимания. Любой желающий узнать, «насколько глубока кроличья нора», может дописать это приложение до конца и выслать мне или опубликовать для продолжения обсуждения. Поверьте, что задача стоит того – если кому-то удастся сходу найти правильное решение – будет интересно услышать его рассказ. В общем случае решение задачи потребует нескольких итераций, хотя внешне задача кажется очень простой.
Всё дело в том, что с этого момента задача перестаёт быть задачей программирования и становится инженерной задачей. Чтобы понять, о чём идёт речь – попробуйте ответить следующие вопросы:

- В каком участке кода (методе) должны рисоваться значения вдоль осей?
- Как нарисовать координатную сетку, отцентрированную относительно осей координат, а не относительно поля рисования? Пояснение: сетка должна быть «привязана» к осям координат – т.е. до ближайшей линии сетки от начала координат вправо и влево должно быть одинаковое расстояние.
- Как обеспечить сосуществование в коде двух режимов рисования – с принудительным переносом осей координат в левый нижний угол и с помещением осей координат в начале координат - с тем, чтобы, исправляя ошибки в работе одного алгоритма рисования, не внести ошибок в работу другого.

Когда будете реализовывать код, особое внимание уделите тестированию в крайних ситуациях. И обеспечивайте внешнее качество результата. Например, значение в начале координат должно быть «0», а не «-0» как это часто бывает.

Мне продолжение темы о решении инженерных задач и рассказе о творческом процессе в режиме рефлексии (также, как написана данная статья) видится очень интересным. Если это интересно не только мне – можем обсудить эту тему, и я продолжу публиковать материалы.

Нам же пора продолжить обсуждение построения модульных интерфейсов.

Шаг N. Сокрытие реализации. Обоюдоострая бритва Оккама.

Сделанное уже выглядит хорошо. Получился достаточно понятный и лёгкий в использовании класс, позволяющий рисовать график. Оставшиеся неприятности незначительны –

пользователь, подключая заголовочный файл, видит детали реализации класса Graphic – его private члены и методы. Хорошо бы избавить пользователя от этих лишних знаний. Он не может получить доступ к этим членам класса – следует ли ему знать о них?

Для решения можно использовать сокрытие реализации или так называемую pimlp-идиому (pimpl – Pointer to IMPLementation).

Решение выглядит следующим образом. Всё определение класса Graphic переносится в файл Graphic.cpp и переименовывается в GraphicImpl (то есть «реализация графика»)

В заголовочном файле остаётся только интерфейс (за исключением одной строки):

```
#pragma once

class Graphic {
  class GraphicImpl* pimpl;
public:
  Graphic();
  ~Graphic();

  // MANIPULATORS
  void ClearContent();
  void AddValue(double x, double y);

  void ShowGrid(bool is_on);
  void AlignAxisToBottomLeft(bool is_on);

  void Draw(CDC *,const CRect& extent);
};
```

Строка

```
class GraphicImpl* pimpl;
```

определяет указатель на объект реального класса, реализующего график (*GraphicImpl*). Класс Graphic в данном случае является лишь оболочкой, перенаправляющей вызовы реальному классу.

В файле исходного текста придётся написать:

```
Graphic::Graphic() {
  pimpl = new GraphicImpl;
}

Graphic::~Graphic() {
  delete pimpl;
}
```

для создания и удаления реального объекта «графика», который будет выполнять всю работу и скрываться за указателем pimpl и методы

```
void Graphic::ClearContent() {
  if (pimpl) pimpl->ClearContent();
}

void Graphic::AddValue(double x, double y) {
  if (pimpl) pimpl->AddValue(x,y);
}
```

и т.д. для каждого метода класса Graphic. Достаточно высокая цена за сокрытие деталей реализации.

Вместо прямого вызова и понятного кода появляются:
- Новый класс-оболочка
- Дополнительные методы перенаправления вызова
- Проверка указателя на реализацию (*pimpl*) на нулевое значение
- Вызов метода через дополнительный указатель
- Создание обекта класса по new в динамической памяти

Стоит ли пользоваться таким тяжёлым решением? Оправдано ли оно?
Оправдано, когда выполняется ряд условий:
- Объекты класса создаются не часто (либо требования по быстродействию и расходу оперативной памяти для нас несущественны).
- Велика вероятность внесения изменений в стуктуру скрываемого класса.

Полная изоляция класса Graphic от пользовалей позволяет вносить любые изменения в структуру класса, не затрагивая при этом другие модули. Их даже не потребуется перекомпилировать.

Очень важным бонусом является то, что теперь заголовочный файл вашего модуля содержит только необходимую вам информацию.

Шаг N+1. Что дальше?

Статья не дописана и оставлена в режиме черновика. В большей степени она ориентирована на новичков в объектно-ориентированном программировании. Для профессионалов она является предметом для обсуждения – правилен ли предлагаемый в статье способ обучения или материал следует строить иначе.

Данная статья не показывает, как должна была решаться целевая задача. Если бы речь шла о решении целевой задачи, то компонент рисования графика следовало взять готовый – в сети достаточно много хороших вариантов.

Статья также не показывает, как правильно формировать код компонента. Если бы речь шла о формировании компонента, то не следовало бы привязывать интерфейс модуля к библиотеке MFC – это затруднит использование компонента в других системах. Следовало бы выбрать наиболее общее решение – WinAPI/GDI.

Статья описывает метод решения задачи, показывает общие приёмы проектирования, учит задавать правильные вопросы и давать на них правильные ответы.

Приглашаю к беседе всех желающих. С радостью принимается любое знание – критика, замечания, опыт использования данной статьи при обучении и самообучении, согласие или несогласие с изложенным в статье. От новичков интересно получить информацию о том, что было непонятно и почему.

Обсуждение можно продолжать на форумах или писать мне email’ом, который можно найти на сайте tenisheff.ru.