Внедрение зависимостей в C/C++

Одной из основных проблем разработки ПО является проблема повторного использования кода. Обычно ее решают путем разбиения программ на бинарные модули или библиотеки, которые могут быть использованы повторно, так как широко используемый, в настоящее время, в системном программировании язык C имеет ограниченные возможности для создания программных компонентов уровня исходных текстов. В частности, затруднено использование немодицифированных исходных текстов из-за их привязки к структуре директорий (относительно которых осуществляется включение заголовочных файлов), к именам включаемых файлов, а также из-за независимости исходных текстов от системы сборки. Если модульность реализована с помощью библиотек, это накладывает ограничения на возможности конфигурирования: оно не может касаться уже скомпилированного кода библиотеки. Если же есть ее исходные тексты, то снова возникает проблема с их использованием в другом проекте без модификаций. В настоящей работе предлагаются методы, позволяющие создавать программные модули уровня исходных текстов (программные компоненты) на языке C, которые могут быть повторно использования без внесения в них каких-либо изменений, а также методы, позволяющие выделять в модулях т.н. сквозную функциональность. При этом используется стандартный инструментарий и синтаксис.

Введение

От структурного программирования к аспектно-ориентированному

Практически сразу после того, как появилась языки программирования, возникла необходимость в структурировании исходных текстов. В конечном итоге это привело к созданию концепции структурного программирования, в котором основная идея была в построении иерархии , а затем и объектно-ориентированного программирования, которое наложило

ограничения на эту иерархию. Вместе с тем, не всякое понятие может быть выражено в терминах классов, объектов и отношений между ними. Существует функциональность, называемая также «сквозной», которая может затрагивать сразу несколько классов или методов. В качестве примера можно привести логирование или поддержку процессов внутри ОС. Класс «процесс» не связан отношениями наследования ни с одной другой сущностью внутри ОС, однако наличие процессов оказывает влияние на нити (которые теперь должны знать, какому процессу они принадлежат), менеджер памяти (который теперь должен переключать адресные пространства) и т.д. То есть, помимо реализации основного класса «процесс», требуются также изменения и внутри других классов, с которыми данный класс не связан отношениями включения или наследования. Иными словами, поддержка процессов требует добавления «процессного аспекта» в нити, менеджер памяти и прочие классы, причем локализовать этот аспект непросто, так как классические ООП-языки требуют полного описания класса в одном месте. Необходимость выделять в модули сквозную функциональность и включение ее в виде одного модуля (который содержит аспекты для всех сущностей, на которые он влияет) привело исследователей из Xerox PARC к созданию в 2001 году концепции аспектно-ориентированного программирования в виде расширения AspectJ для ООП-языка Java. Впоследствии появились также расширения и для других языков, в том числе для C++. В основе концепции АОП лежит возможность описывать некоторую сквозную функциональность (аспект) в виде набора «советов» (advice). «Советом» может быть, например, вызов функции логирования. С каждым «советом» связано описание «точки соединения» (join point), куда следует добавить «совет». Поскольку весь аспект может быть описан в одном месте, это решает проблему включения сквозной функциональности: удалив аспектный модуль, удаляются все «советы» из всех мест, где они применялись. Поскольку описание такой точки является частью описания совета, целевые модули не содержат никакой информации о внедряемом аспекте. В качестве точек соединения используется определенный набор языковых конструкций, например начало или конец определенной функции, все вызовы какого-либо метода и т.д.

Возможность использования только определенных синтаксических конструкций в качестве точек соединения и является основным ограничением АОП, так как логика аспекта может быть не привязана к этим конструкциям. Кроме этого, возникают ложные зависимости, поскольку аспект привязывается к определенным синтаксическим конструкциями внутри других модулей, использование аспекта с другими модулями затрудняется. Попытки разрешить эти трудности привели к созданию технологии внедрения зависимостей (dependency injection). В классическом ООП, несмотря на отделение интерфейса от реализации, фактически текст программы все равно содержит описание того, какие реализации используются для определенных интерфейсов. Идея заключается в том, чтобы вынести эту информацию из текста программы, в некоторые метаданные, хранящие зависимости. Изменяя эти метаданные, можно изменять используемые реализации интерфейсов без изменения текста программы. Следует сказать, что здесь и далее обсуждается только статические программы, если решение об использовании определенного интерфейса откладывается до времени выполнения, то, хотя это и решает частично проблемы зависимостей, может быть неприемлемо по соображениям производительности. Несмотря на то, что АОП является парадигмой программирования, ВЗ является шаблоном проектирования, то есть формально может быть реализовано в любом языке. В реальности, в любом языке может быть реализовано только ВЗ времени выполнения, для статического ВЗ (когда связывание компонентов осуществляется на этапе компиляции) нужна определенная поддержка со стороны языка, и статическое ВЗ может быть реализовано в рамках не всякого синтаксиса. К счастью, C и C++ обладают необходимыми возможностями, позволяющими без изменения синтаксиса реализовать статическое ВЗ.

Хотя АОП и внедрение зависимостей (ВЗ) часто называют конкурирующими технологиями, на самом деле они ни в чем не противоречат друг другу и призваны решать разные задачи. Аспектно-ориентированное программирование предполагает, что точки соединения указываются в аспекте, то есть разработчик аспекта пишет, где именно должен быть применен данный код, притом что сам изменяемый код об этом не знает. Этот подход

хорош для простых задач, вроде логирования, когда нужно что-то добавить в каждый метод или во все определенные точки. Если же надо внедрить аспект в несколько специальных точек, то задача становится нетривиальной, т.к. возникает недостаток информации – целевой код не содержит никаких данных о точках, где может быть внедрен аспект, но набор «стандартных» точек не соответствует нужному набору. Напротив, внедрение зависимостей предполагает, что точки соединения описаны в целевом коде (в виде вызова функций, макросов и т.д.), поэтому задача сводится к тому, что необходимо включить нужные вызовы или макросы в качестве «советов». Иными словами, ВЗ должно использоваться в ситуации, когда есть интерфейс, и требуется возможность использования различных реализаций, причем в тексте программы не должно быть никаких упоминаний этих реализаций и они могут не знать, кто будет их использовать. АОП же скорее призвано выделять в модули сквозную функциональность для конкретных программ.

Методы конфигурирования программ на С

Цель данной работы – исследовать возможности использования внедрения завимостей для создания статических программных компонентов на языке С (которые могут быть без модификаций использованы в различных проектах), а также возможности конфигурирования (т.к. с помощью изменения реализаций интерфейсов можно добиться изменения свойств системы). Обсуждение ведется в контексте встраиваемых систем, так как важным отличием последних являются повышенные требования к оптимальности получаемого кода. Если для персональных компьютеров и ОС общего назначения допустимо использование динамического полиморфизма для изменения свойств программ, то для встраиваемой системы это может привести к проблемам с производительностью.

Классическим способом конфигурирования программ на С является использование условной компиляции. Во всех точках программы, поведение которых может меняться, находится директива #ifdef:

```
#ifdef CONFIG_A

Code1…

#elif CONFIG_B

Code2…

#elif CONFIG_C

Code3…

#endif
```

Также распространен вариант, когда в коде находится вызов функции, а реализация этой функции помещена в блок #ifdef:

```
#ifdef CONFIG_A

Func{

 Code1…

}

#endif


#ifdef CONFIG_B

Func{
```

```
 Code2…
}
#endif


#ifdef CONFIG_C
Func{
 Code3…
}
#endif
```

В этом случае, несмотря на наличие нескольких реализаций функции, только одна из них попадет в результирующий объектный файл, поэтому код, использующий эту функцию, может просто вызвать ее, во время компоновки этот вызов будет связан с нужным экземпляром функции.

Наконец, можно вообще убрать все директивы условной компиляции, и использовать систему сборки для того, чтобы только один из файлов, содержащих реализацию функции Func, был использован компоновщиком. Однако при этом могут возникнуть накладные расходы времени выполнения, если нужно отключить данный функционал (исключить вызов Func), так как объектные файлы уже содержат вызов функции, какая-то функция все равно должна быть вызвана (возможно функция-

заглушка).
Первый вариант хорош тем, что конфигурирование осуществляется на уровне исходных текстов, и, таким образом, не приводит к накладным расходам, код всегда содержит ровно то, что нужно в каждом конкретном случае. Из недостатков можно отметить то, что такой программный модуль является «закрытым» для внешнего конфигурирования: если появляется еще один вариант конфигурации, это потребует изменения всех компонентов, которые его используют (дополнить их еще одним вариантом #ifdef). Таким образом, система представляет собой черный ящик, где конфигурирование возможно только в пределах, изначально предусмотренных разработчиком модуля.
Второй и третий варианты позволяют добавить сторонний код как аспект, но при этом имеют накладные расходы во время выполнения, в случае если аспект необходимо отключить. Было бы желательно избавиться от недостатков этих способов, сохранив при этом их достоинства.

Существующие технологии АОП и ВЗ для С

Нельзя сказать, что вышеописанные проблемы не пытались решать. Прежде чем описывать предлагаемую технику, будут рассмотрены существующие разработки, предназначенные для решения этих проблем.

Аспектные препроцессоры для C/C++

После создания в 2001 году AspectJ для Java, предпринимались попытки создать нечто похожее и для C/C++. После адаптации синтаксиса под особенности этих языков были разработаны аспектные препроцессоры AspectC, AspectC, FeatureC+ и т.д. Все эти технологии имеют один существенный недостаток: они, по-сути, являются другим языком, надстроенным над C++, который требует отдельного изучения, а также такой код невозможно собрать, не имея соответствующего компилятора/препроцессора. Кроме того, как уже обсуждалось выше, набора стандартных точек входа, предоставляемых АОП, может быть недостаточно для описания требуемого аспекта.

Ссылки файловой системы

В некоторых ОС, например в Linux, эта проблема обходится с помощью возможностей файловой системы, вместо файла создается ссылка (имеющая некоторое абстрактное имя), далее эта ссылка отображается на один из конкретных файлов, после чего директива #include использует этот конкретный файл, хотя в самом исходном тексте имя указано абстрактно. Например, вместо того чтобы писать в программе

#include <x86.h>

или

#include <arm.h>

в тексте программы написано

#include <cpu.h>

где имя файла является ссылкой, которая должна быть настроена на нужный файл с помощью средств файловой системы, в зависимости от того, под какой процессор производится сборка.

Данный метод решает все описанные проблемы:

1. Легкая возможность добавления новых аспектов (ссылка может быть настроена на любой другой файл без изменения текста программы.
2. Отсутствие накладных расходов (при необходимости, можно написать файл-заглушку, который вместо прототипов функций будет содержать «пустые» макросы, и таким образом, вызовы функций будут исключены из исходного текста препроцессором и не будут видны компилятору.
3. Полная совместимость со стандартом языка, не вводится никаких изменений синтаксиса, не требуется никаких внешних инструментов.

Из недостатков данного метода можно отметить только то, что не все ОС поддерживают создание ссылок. Учитывая, что с помощью данного метода достигаются все поставленные цели, можно заключить, что для реализации ведрения зависимостей необходимо найти способ включения абстрактных интерфейсов в качестве заголовочных файлов, и возможность указывать

соответствие между этим интерфейсом и конкретным файлом извне, не изменяя код программы (по аналогии с созданием ссылок).

Способ внедрения зависимостей

Используемые методы внедрения зависимостей основаны на концепции аннотаций, или метаданных, которые были подробно рассмотрены в статье «Аннотации как средство управления сложностью».

Прежде чем рассматривать методы внедрения зависимостей, введем несколько оределений.

Совокупность типов данных, прототипов функций, определений и прочей информации, которая может быть расположена в заголовочном файле называется его **интерфейсом**. Нужно отметить, что под это определение подпадают только те сущности, которые могут быть использованы пользователем данного заголовочного файла (документированные). Внутренние определения, и функции внутри заголовочного файла интерфейсом не являются.

Связанные с заголовочными файлами исходные тексты, содержащие реализацию функций описанных в соответствующих заголовочных файлах называются **реализацией интерфейса** или просто **реализацией**.

Зависимостью модуля А от модуля Б называется включение заголовочного файла модуля Б в заголовочный файл или файл исходного текста модуля А, так как это предполагает использование интерфейса модуля Б внутри интерфейса или реализации модуля А. Использование функций, макросов и прочих элементов их модуля Б внутри модуля А формирует, таким образом, набор точек соединения или срез.

С помощью метаданных группы файлов объединяются в модули, которые содержат в виде метаданных информацию об импортируемых и экспортируемых интерфейсах.

Имена интерфейсов указываются в следующей форме

FX_METADATA(({ interface: [INTERFACE, IMPLEMENTATION] }))

Где INTERFACE - это имя интерфейса (абстрактного набора функций и типов данных, которые могут быть использованы после

включения данного заголовочного файла), а IMPLEMENTATION – имя данной реализации (обычно, это версия, либо нечто, отличающее данную рализацию от других), поскольку интерфейс может иметь множество реализаций, необходимо как-то различать их между собой. Важно отметить, что заголовочный файл также является частью реализации, поскольку один и тот же набор функций и макросов может описываться разными заголовочными файлами.

В свою очередь, соответствующий данному интерфейсу файл исходного текста, должен содержать в себе строку :

FX_METADATA(({ implementation: [INTERFACE, IMPLEMENTATION] }))

Отношения между файлами и интерфейсами выглядят следующим образом: один заголовочный файл может иметь только один элемент interface с именем интерфейса. Интерфейс может иметь 0 или более реализаций (если не существует файла с соответствующей меткой implementation, считается, что интерфейс полностью описан в заголовочном файле). Хотя интерфейс может не иметь файла реализации, файл реализации обязан иметь интерфейсный заголовочный файл. Это требование вытекает из определения зависимости интерфейсов, если реализация не содержит интерфейса, следовательно, она не может быть использована директивой include, следовательно, ее невозможно использовать.

Отображение

Исходя из целей и задач внедрения зависимостей, информация о зависимостях не может храниться внутри исходных текстов, поэтому она хранится в виде некоторых внешних данных, так называемого **отображения**. Отображение (или карта) интерфейсов содержит строки вида:

INTERFACE = VERSION1

которые говорят о том, что в качестве интерфейса INTERFACE по-умолчанию следует использовать реализацию VERSION1. Везде, где используется указанный интерфейс, следует включать файл, содержащий метку [INTERFACE, VERSION1], а все файлы реализации этого интерфейса должны быть включены в списки, которые попадут к компоновщику.

Включение интерфейса

Другим важным вопросом является включение интерфейса по его имени, а не по имени файла, как это делается в языке C. Для этой цели используется оговоренная в стандарте возможность писать макросы в качестве аргумента директивы #include. Создается специальный макрос FX_INTERFACE, который принимает имя интерфейса в качестве аргумента. Например, включение мьютексов в прикладную программу должно выглядеть так:

#include FX_INTERFACE(MUTEX)

Этот макрос, а также макросы, соответствующие именам файлов, должны быть включены неявно директивой компилятора (force include), либо переданы в качестве ключа командной строки. Способы определения этого макроса, а также отображения имен интерфейсов на имена файлов подробно рассмотрены в статье, посвященной аннотациям в C.

Внедрение зависимостей

Внедрение зависимостей – процесс сопоставления абстрактным интерфейсам их реализаций. В более узком смысле – процесс сопоставления абстрактным включениям вида :

#include FX_INTERFACE(MUTEX)

соответствующего заголовочного файла, содержащего нужный интерфейс, тот, который указан в метаданных, например, если в последних указано, что

MUTEX = VERSION1

То следует включить заголовочный файл, содержащий метку

FX_METADATA(({ interface: [MUTEX, VERSION1] }))

Так как, в конечном итоге, имена интерфейсов отображаются на имена заголовочных файлов с помощью #define, и, в свою очередь, #include использует такие же макросы, достаточно просто создать определения, отображающие одно на другое. Например, после анализа исходных текстов, каждый файл будет иметь доступ к определению вида

#define MUTEX__VERSION1 "c:\mutex.h"

Такая строчка там появится после анализа файла mutex.h, содержащего метку интерфейса [MUTEX, VERSION1].

Затем, после анализа файла отображений там появится

отображение:

#define MUTEX MUTEX__VERSION1

Теперь, директива #include FX_INTERFACE(MUTEX) приведет к включению файла mutex.h. Если граф зависимостей требует включения данного модуля, то в список файлов подлежащих компиляции попадут также все файлы, содержащие метку реализации данного интерфейса.

Отслеживание зависимостей

Поскольку вся информация о зависимостях интерфейсов друг от друга содержится в самих исходных текстах, было бы желательно найти способ извлечь ее оттуда, не прибегая к ручному описанию зависимостей интерфейсов друг от друга. Данная задача может быть решена стандартными средствами. После того как был сформирован файл отображений и известно, какие реализации каким интерфейсам соответствуют, можно использовать стандартный препроцессор C для извлечения информации о зависимостях. Поскольку в результате работы препроцессора в целевой файл включатся все заголовочные файлы, которые он импортирует, а в каждом заголовочном файле имеется метка, то все, что нужно сделать – это извлечь из вывода препроцессора метки interface и implementation. Все найденные метки реализации зависят от всех найденных меток интерфейса. Это извлечение может быть произведено с помощью простых регулярных выражений, т.к. если известно, что метаданные записаны внутри специфического макроса, можно определить его таким образом, чтобы перед и после метаданных вставлялась определенная метка, позволяющая точно определить границы метаданных в тексте программы.

В случае если отображение не готово заранее (а следовательно, файлы не могут быть обработаны препроцессором из-за неизвестности имен файлов для include), импортируемые интерфейсы могут быть извлечены путем ручного разбора содержимого директив include, либо с помощью дополнительных возможностей используемых компиляторов для анализа зависимостей, которые также были рассмотрены в статье, посвященной аннотациям.

Итог

Для реализации внедрения зависимостей, можно использовать внутрифайловые метаданные.
Вместо включения заголовочных файлов по имени, необходимо использовать включение интерфейса с помощью макроса FX_INTERFACE. Кроме того, поскольку становится возможно извлечь из самих файлов информацию о зависимостях, отпадает необходимость ручного описания зависимостей (например, в makefiles). Из файлов исходных текстов и файла отображений всегда можно сказать, какие файлы необходимо скомпилировать, для того чтобы реализовать указанный интерфейс и все, от чего он зависит.

Процесс обработки исходных текстов

Вначале, из заголовочных файлов извлекается информация об экспортируемых ими интерфейсах. Для этого все заголовочные файлы, содержащиеся в целевых папках, просматриваются на предмет наличия в них меток интерфейсов. Список этих папок указывается в качестве входных данных для внедрения зависимостей. Затем генерируется файл отображений имен интерфейсов на имена файлов, а также отображение абстрактных интерфейсов на конкретные. После того, как получен этот файл, может быть построен граф зависимостей, поскольку становятся известны реализации всех интерфейсов и их зависимости. Данный граф может быть построен автоматически, он хранится в виде отображения имени интерфейса, на его зависимости. Наконец, имея имя целевого интерфейса, можно рекурсивно разрешать зависимости, параллельно запоминая файлы исходных текстов, которые реализуют эти интерфейсы. После того как граф будет обработан, на выходе получается список файлов, которые должны быть скомпилированы, а также заголовочный файл, содержащий информацию о внедренных зависимостях.

Пример

В качестве простого примера, можно привести часто возникающую на практике ситуацию, когда в коде необходимо

включать и выключать логирование, а также поддерживать множество возможных реализаций этого логирования (вывод в COM-порт, на консоль и т.д.), причем не только тех, которые предусмотрены в коде, но и тех, которые могут быть добавлены в будущем. Исходные тексты при этом меняться не должны.

Конфигурирование с помощью традиционных способов, описанных в разделе «Цели» не позволяет одновременно разрешить все поставленные вопросы. Посмотрим, как эту проблему можно решить с помощью внедрения зависимостей.

В качестве файлов, использующих логирование, будем использовать file1.c и file2.c. В качестве заголовочного файла и реализации логирования, будем использовать пары log1.h/log1.c, log2.h/log2.c и так далее, файл, содержащий точку входа в программу назовем main.c.

В качестве интерфейса логирования, используется функция simple_log, имеющая следующий прототип:

void simple_log(char* log_msg);

Содержимое заголовочного файла log1.h, который должен выводить лог в COM-порт:

```
#ifndef SIMPLE_LOG_COM_PORT

#define SIMPLE_LOG_COM_PORT



void simple_log_to_com_port(char* log_msg);

#define simple_log(s) simple_log_to_com_port(s)



FX_METADATA(({ interface: [SIMPLE_LOG, COM_PORT] }))

#endif
```

Соответствующий ему файл реализации log1.c:

```
FX_METADATA(({ implementation: [SIMPLE_LOG,COM_PORT] }))

void simple_log_to_com_port(char* log_msg)
{
  // output to RS232C interface.
}
```

Содержимое заголовочного файла log2.h, который должен выводить лог в консоль:

```
#ifndef SIMPLE_LOG_CONSOLE
#define SIMPLE_LOG_CONSOLE

void simple_log_to_console(char* log_msg);
```

```
#define simple_log(s) simple_log_to_console(s)


FX_METADATA(({ interface: [SIMPLE_LOG, CONSOLE] }))

#endif
```

Соответствующий ему файл реализации log2.c:

```
FX_METADATA(({ implementation: [SIMPLE_LOG, CONSOLE] }))


void simple_log_to_console(char* log_msg)

{

  printf(log_msg);

}
```

Файл-заглушку, который отключает логирование (log3.h) можно написать так:

```
#define simple_log(ignored)

FX_METADATA({ interface: [SIMPLE_LOG, STUB] })
```

Файла file1.c, использующий логирование, использует его в абстрактной форме:

```
#include FX_INTERFACE(SIMPLE_LOG)

FX_METADATA({ implementation: [MODULE1, VER1] })


void func_with_log1(void)

{

  simple_log("hello, I'm log from file1!");

}
```

Содержимое файла file2.c

```
#include FX_INTERFACE(SIMPLE_LOG)

FX_METADATA(({ implementation: [MODULE2, VER1] }))


void func_with_log2(void)

{

  simple_log("hello, I'm log from file2!");

}
```

Заголовочные файлы для file1.c и file2.c содержат прототип функций и метку соответствующего интерфейса.
Наконец, главный файл выглядит так (заголовочный файл должен содержать просто метку интерфейса):

```
#include FX_INTERFACE(MODULE1)

#include FX_INTERFACE(MODULE2)

FX_METADATA(({ implementation: [MAIN, VER1] }))


void main(void)

{
```

```
  func_with_log1();

  func_with_log2();

}
```

Рассмотрим детально процесс работы данного инструмента:
На первом этапе извлекаются все заголовочные файлы, лежащие в указанных директориях, и в них ищутся метки интерфейсов, в результате создается следующий список:
MAIN:VER1 – main.h
MODULE1:VER1 – file1.h
MODULE2:VER1 – file2.h
SIMPLE_LOG:COM_PORT – log1.h
SIMPLE_LOG:CONSOLE – log2.h
SIMPLE_LOG:STUB – log3.h
затем обрабатывается файл отображений, предположим, что логированием «по-умолчанию» является вывод в консоль, то есть отображение содержит строки:
SIMPLE_LOG = CONSOLE
MODULE1 = VER1
MODULE2 =VER1
После этого обрабатываются файлы исходных текстов (в них ищутся метки реализаций) и получается следующая структура:

Интерфейс (ключ)	Зависимости	Файлы реализации
MAIN:VER1	MODULE1:VER1 MODULE2:VER1 SIMPLE_LOG:CONSOLE	main.c
MODULE1:VER1	SIMPLE_LOG:CONSOLE	file1.c

MODULE2:VER1 SIMPLE_LOG: CONSOLE file2.c
SIMPLE_LOG: CONSOLE log1.c

Зависимость интерфейса MAIN:VER1 от SIMPLE_LOG, хотя и отсутствует в исходных текстах, зато появляется после того как построен полный граф зависимостей, т.к. этот интерфейс включается косвенно.

Выполняется рекурсивный обход графа, начиная с MAIN:VER1, и выяснение всех элементов, которые реализуют все нужные интерфейсы, на выходе получается файл header, содержащий информацию о заголовочных файлах и внедренные зависимости:

```
#define SIMPLE_LOG__COM_PORT "c:\my_program\log_com\log1.h"
#define SIMPLE_LOG__CONSOLE "c:\my_program\log_con\log2.h"
#define SIMPLE_LOG__STUB "c:\my_program\log_stub\log3.h"
#define MODULE1__VER1 "c:\my_program\file1.h"
#define MODULE2__VER1 "c:\my_program\file2.h"
#define MAIN__VER1 "c:\my_program\main.h"
#define SIMPLE_LOG SIMPLE_LOG__COM_PORT
#define MODULE1 MODULE1__VER1
#define MODULE2 MODULE2__VER1
```

И список файлов, подлежащих компиляции:

```
C:\my_program\main.c
C:\my_program\file1.c
C:\my_program\file2.c
C:\my_program\log_con\log2.c
```

Во время обхода графа, после обработки узла, он помечается как уже обработанный, это исключает повторную обработку одного и того же интерфейса, а также зацикливание механизма извлечения зависимостей: имея конечное число элементов n, граф

будет гарантированно обработан за конечное время O(n). При этом, поскольку извлечение зависимостей начинается с корня, указанного пользователем, интерфейсы, к которым нет путей из этого корня (от которых он не зависит) не будут обработаны и не войдут в результирующий образ.

Теперь, с помощью изменения отображений интерфейсов, можно легко менять реализацию интерфейса логирования или отключать его. Данные программные компоненты могут быть легко использованы в сторонних проектах, они не зависят от структуры директорий и конфигурируются во время компиляции.

Ограничения

Так как язык C не поддерживает пространства имен, все функции существуют в глобальном пространстве, а если функция является частью интерфейса, то это означает, что интерфейс может быть определен только для всей системы в целом (то есть является синглтоном (singleton)). Если в систему входят несколько модулей, каждый из которых импортирует определенный абстрактный интерфейс, при определении конкретного интерфейса он может быть определен только для всех модулей сразу.

Для того, чтобы найти все метки интерфейсов, нужно открыть и просмотреть все файлы, присутствующие в указанных папках, что, для больших проектов, может приводить к большим затратам времени.

Будущая работа

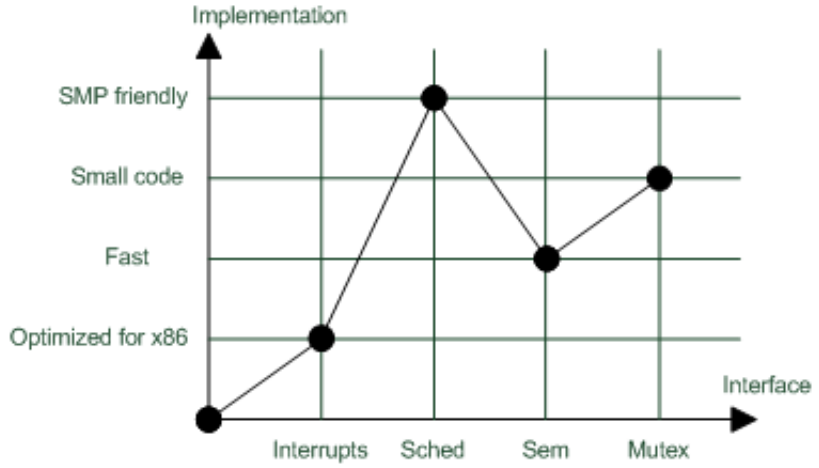
Автоматическое конфигурирование

Во время анализа и внедрения зависимостей, может оказаться, что импортируемый интерфейс имеет только одну возможную реализацию, это означает, что нет никакой неоднозначности в выборе, какой именно интерфейс использовать в качестве интерфейса по-умолчанию, поэтому файл отображений может быть сгенерирован автоматически, на основании только списка путей к исходным файлам.

Взвешенные интерфейсы и системные профили

Автоматическая конфигурация, рассмотренная в данном разделе позволяет генерировать файл отображений, используя отсутствие альтернативы. Если есть несколько реализаций интерфейса, требуется внешнее вмешательство, для того чтобы выбрать одну из них. Развитие идеи автоматического конфигурирования предполагает введение дополнительных атрибутов, по которыми могут различаться интерфейсы, тогда интерфейс имеет «вес» и может быть выбран автоматически из нескольких альтернатив. Например, интерфейсы могут иметь некоторые свойства, такие как «пригоден для многопроцессорных систем», «имеет наименьший размер кода» и так далее, тогда, указав приоритетные свойства, становится возможным по набору путей к исходным файлам автоматически построить конфигурацию, которая в наибольшей степени соответствует этим свойствам. Таким образом, возможно создание профилей системы, имеющей одинаковый функционал по набору предоставляемых функций, но различающихся по свойствам реализаций этих интерфейсов.

Говоря формально, если по оси x перечисляются возможные интерфейсы, а по оси y – их возможные реализации (перечисленные в порядке их приоритетности) то речь идет об отыскании функции профиля, которая дает оптимальную реализацию (существующую и имеющую оптимальный «вес» для данного набора критериев) для указанного интерфейса x. С помощью этой функции, может быть построена таблица отображений интерфейсов на реализции и осуществлено внедрение зависимостей и сборка системы.



Бинарные реализации

Все, что обсуждалось до сих пор, касалось исключительно исходных текстов. Вместе с тем, хотя интерфейс и определяется в заголовочном файле, его реализация может и не быть исходным текстом (может быть уже скомпилирована). При этом теряется возможность конфигурирования, но это может быть неважно (если, например, интерфейс не предполагает такого конфигурирования). Требуется возможность рассматривать объектные файлы или библиотеки как реализации интерфейсов и их включение в списки компоновщика минуя стадию компиляции.

Заключение

Внедрение зависимостей имеет большие перспективы для конфигурирования встраиваемого ПО, так как дает возможность разработки программных компонентов, которые могут конфигурироваться на уровне исходных текстов, и таким образом, не содержат накладных расходов времени выполнения. Использование внедрения зависимостей позволяет решить сразу несколько задач, связанных с поддержкой исходных текстов: получаемая система получается «открытой» и допускает добавление в себя новой функциональности без изменения исходных текстов, кроме того, компоненты не зависят от структуры папок и могут быть повторно использованы в проектах с любой структурой, не требуется ручного описания зависимостей интерфейсов между собой, так как эта информация может быть извлечена автоматически из исходных текстов программы.

Также есть база для разработки в будущем инструментов, позволяющих осуществлять полностью автоматическое конфигурирование и определение входящих в систему модулей на основании информации о требуемых пользовательскому приложению интерфейсов, а также требований к реализации этих интерфейсов.