Повторное использование против сжатия *

Ричард П. Гэбриель
перевод Евгения Охотникова †

Oxford University Press 1996

Перевод 2006

Может быть это моя плохая память, но мне представляется, что крючком, который подцепил весь мейнстрим и привел его к объектно-ориентированному программированию, было “повторное использование кода”. Одной из самых сложных задач, которая есть у руководителей разработки — это как быстрее выполнять большие программные проекты. Большинство людей согласны, что сопровождение — это значительная часть общей сложности программного обеспечения (ПО), но для организаций, чье выживание зависит от выпуска новых программ или продуктов, важнейшим вопросом является именно выпуск нового ПО.

Внутри каждой организации, пишущей большое количество программ — и среди множества организаций пишущих большое количество программ — кажется, что значительная часть программного обеспечения должна быть повторно используемой. Если бы это было так, то можно было взять часть уже существующего кода и поместить его в новый продукт, тем самым, уменьшая время производства желаемого ПО. Более того, если код повторно использован, то очень возможно, что он хорошо протестирован и, вероятно, не содержит багов, но даже если это не так, то все равно сопровождение различных программ, содержащих повторно используемый код, должно быть проще.

Повторное использование — это не новая идея. Десятилетиями языки имели поддержку понятия библиотек. Библиотека — это набор подпрограмм, обычно из какой-то прикладной области. Старыми примерами являются научные библиотеки подпрограмм для Фортрана. Похожей идеей были “Накопленные Алгоритмы” опубликованные ACM много лет назад в виде кода на ALGOL-подобном языке. Я помню, как молодым человеком в 1968 году искал алгоритмы сортировок и поиска на своей первой работе программиста.

Тем не менее, каждый руководитель знает, что повторное использование при этих обстоятельствах требует организованного процесса повторного использования или, как минимум, политики. Во-первых, вам нужно иметь централизованный репозиторий кода. Но он не поможет, если разработчики вынуждены постоянно обращаться к другим разработчикам в поиске кода, который вы можете использовать повторно. Некоторые организации достаточно маленькие и программисты могут устраивать групповые встречи для обсуждения нужного им кода и его производства.

*Это перевод главы Reuse Versus Compression из книги Ричарда Гэбриеля “Patterns of Software. Tales from the Software Community” [1]

†Огромная благодарность Ирине Дегтяревой за помощь в переводе и за усилия, которые она тратит на обучение английскому языку нерадивых учеников вроде меня.

Во-вторых, необходимо иметь средства обнаружения подходящих фрагментов кода, которые обычно требуют хорошей схемы классификации. Нет ничего хорошего в наличии подходящего фрагмента кода, если никто не может найти его. Классификация в мире книг, отчетов, научных журналов и т.п. — это профессия, называемая *каталогизация*. Библиотекари помогают людям находить книги. Но немногие организации могут позволить себе каталогизатора ПО, не говоря уже о библиотекаре который помогал бы разработчикам в поиске программного обеспечения. Это потому, что когда руководитель разработки имеет выбор между наймом еще одного разработчика или библиотекаря руководитель всегда нанимает разработчика. Хорошо бы посмотреть, почему это так и я вернусь к этому позднее.

В-третьих, должна быть хорошая документация на то, что делает каждый находящийся в репозитории фрагмент кода. Включающая не только интерфейс и его назначение, но также достаточно о внутренностях кода — его производительности и использовании ресурсов — чтобы позволить разработчикам широко его использовать. Разработчик должен знать эти вещи, например, чтобы удовлетворить требованиям производительности. Во многих случаях такая документация - это код сам по себе, но подобная информация все же лучше представляется обычной документацией; однако вновь, руководитель разработки предпочтет нанять разработчика, а не человека для поддержки документации.

Многие мелкие фрагменты кода гораздо проще написать самостоятельно, чем проходить через процесс нахождения и изучения повторно используемого кода. Следовательно, руководители обнаруживают, что ориентированный на процесс мир повторного использования имеет слишком много барьеров для эффективного применения.

Глядя на природу этого подхода к повторному использованию мы можем видеть, что он больше сфокусирован на повтором использовании кода из одного проекта в другом проекте, чем внутри одного проекта; а процесс повторного использования охватывает всю организацию.

Обратимся к объектно-ориентированному программированию. Основной пункт объектно-ориентированного программирования — это смещение фокуса в проектировании программ с алгоритмов на структуры данных. В частности, структуры данных достигают точки, в которой они могут содержать собственные операции.

Но такие структуры данных, называемые *объектами*, часто соотносятся друг с другом особенным способом: один является почти таким же, как второй, за исключением некоторых добавлений или незначительных модификаций. В такой ситуации, если эти объекты полностью независимы, оказывается настолько много дублирования, что было разработано средство наследования кода — *методы*.

Поэтому мы видим притязание на повторное использование в объектно-ориентированных языках: когда пишется отдельная программа программист повторно использует уже разработанный код, наследуя его из более общих объектов или классов. В этом красота такого рода повторного использования: оно не требует особого процесса, поскольку является частью природы языка.

Коротко говоря, *создание подклассов* является средством повторного использования.

Несмотря на подобный упрощенный взгляд на повторное использование, идея о том, что в объектно-ориентированных языках повторное использование является частью их сущности, подтверждается большим вниманием со стороны мейнстримового сообщества. Определенно есть и другие притягательные качества, вот некоторые из них:

- объекты и классы объектов для программистов являются естественным путем ор-

ганизации программ;

- системы, написанные в объектах и классах проще расширять и подстраивать чем разработанные традиционными способами.

* * *

Тем не менее, форма повторного использования в объектно-ориентированных языках едва ли удовлетворяет обширным целям разработки программного обеспечения. Что я хочу предложить — это более подходящий термин, чем *повторное использование* и, может быть, лучшую концепцию для того качества объектно-ориентированных языков, которое выглядит как *повторное использование*.

Термином (и концепцией) является *сжатие*. Сжатие — это такое качество фрагмента текста, что смысл любого из его кусочков *больше* чем у самого фрагмента. Это достигается глубоким контекстом и каждая часть текста формируется этим контекстом — каждое слово черпает часть смысла из своего окружения. Хорошо знакомым примером, не связанным с программированием, является поэзия, в которой сложное значение может казаться чрезвычайно насыщенным из-за множества производимых ей образов и способом, посредством которого каждый новый образ или фраза вырисовывается из нескольких других. Поэзия использует сжатый язык.

Вот однострочный пример: *My horse was hungry, so I filled a morat with oats and put it on him to eat*.

Эта фраза сжата достаточно для того, чтобы странное слово *morat* стало очевидным — это мешок для корма. Местоимения работают благодаря сжатию.

Сжатие в объектно-ориентированных языках возникает, когда определение подкласса берет значительную часть своего значения из определений своих суперклассов. Если вы создаете подкласс, который добавляет к классу один атрибут и два метода, описанием этого нового класса будет всего лишь новый атрибут, два метода и ссылка на уже существующий класс. Может казаться, что программист, написав подкласс, повторно использует код суперкласса, но лучше смотреть на это так, что программист пишет сжатое определение, масса деталей которого берется из контекста в котором суперкласс уже существует.

Для того, чтобы убедиться в этом, отметьте, что определение подкласса часто находится на некотором расстоянии от определения суперкласса, поэтому программист полагается на знание о суперклассе для написания короткого определения.

Сжатый код имеет одно интересное свойство: чем больше его вы пишите, чем дальше по иерархии классов вы опускаетесь, тем более сжатым становится новый код. Это хорошо, по крайней мере, по одной причине: добавление нового кода выполняется довольно компактно. Для руководителя разработки это *означает*, что новый код может быть написан быстрее.

Здесь нам может помочь смена точки зрения: сжатие имеет явные недостатки и эти недостатки могут объяснить, почему объектно-ориентированные языки не решили проблем программного обеспечения (естественно, существует множество причин, но, в конце-концов, это всего лишь первое эссе).

Отчасти сжатие опасно потому, что оно требует от программиста глубокого понимания контекста, из которого сжатый код берет свое значение. Для чего необходим не только доступный исходный код или отличная документации, но и сама природа наследованного языка принуждает программиста к пониманию исходного кода или документации. Если программист нуждается в значительной части контекста для понимания

программы, которую он должен расширить, то он может совершить ошибку просто из-за недопонимания.

Более того, даже если новый код — сжатый код — компактен, он может потребовать, как минимум, столько же времени и сил, сколько понадобилось бы для написания несжатого кода, за исключением случая, когда общая сложность или общий объем кода невелики, или когда пишущий его человек полностью удерживает существующий код в памяти.

Сопровождение сжатого кода требует понимания его контекста, что может быть тяжело. Основной чертой легкого сопровождения является *локальность*: локальность — это такая характеристика исходного кода, которая позволяет программисту понять исходник только по его небольшому фрагменту. Сжатый код не обладает таким качеством, если только вы не пользуетесь какой-либо очень замечательной средой разработки.

Сжатие имеет еще одну очевидную проблему. Если вы строите производный код, который сильно зависит от какого-то базового кода, то изменение базового кода может быть дорогим и опасным. Если разработчик базового кода не является разработчиком производного кода, то производный код может оказаться в опасности. Это просто проблема связи всех со всеми. Сжатие в поэзии — это хорошо, поскольку первоначальные определения слов и фраз находятся вне мыслей поэта. Это не так по отношению к сжатию в программах: значения слов — классов — определяется программистом. Эта проблема не уникальна для основанного на классах сжатия, она характерна для любого сжатия, основанного на абстракциях.

Я не хочу намекать, что сжатие — это плохо, в действительности это очень важный ресурс для итерационного определения, но на пути могут возникнуть общие проблемы, такие как рискованность изменения базового кода.

Сжатие не может помочь повторному использованию между проектами, поскольку это требует экспорта классов, которые могли бы использоваться повторно. Что ведет нас к классической ситуации с повторным использованием, которую мы имели ранее, где основным барьером является процесс разработки, а не технология.

* * *

Помните, я указывал, что руководители разработки всегда нанимают разработчиков вместо тех, чьей работой является повышение эффективности повторного использования. Почему это так?

Некоторые причины должны быть очевидны. Во-первых, повторное использование проще внутри одного проекта, чем между несколькими. Успех руководителя зависит от скорости выполнения конкретного проекта, а нет от общей скорости выполнения ряда проектов. А подготовка кода к повторному использованию требует дополнительной работы, не только от специалистов по повторному использованию, но так же и от разработчиков. Поэтому подготовка к повторному использованию накладывает дополнительную стоимость на каждый конкретный проект.

В конце концов, успех проекта зависит, как минимум частично, от производительности разрабатываемого кода. Руководитель разработки знает, что большая часть разрабатываемого кода отличается от уже имеющегося кода лишь потому, что он специализируется под частные случаи, чтобы быть достаточно быстрым. Поэтому в данной ситуации повторное использование не может оказать значительной помощи.

Мы увидели, что повторное использование — это, по сути, вопрос организации разработки и каждой организации необходимо определиться, верит ли она в долговременную выгоду. Объектно-ориентированные языки предоставляют сжатие, которое может служить заменой повторному использованию внутри программы, но за счет стоимости

сопровождения.

Сжатие — это не повторное использование и у обоих, повторного использования и сжатия, есть своя цена и своя экономия.

Список литературы

[1] *Richard P. Gabriel*, Patterns of Software Tales from the Software Community, Oxford University Press, 1996. *Доступна в виде PDF:* `http://www.dreamsongs.com/NewFiles/PatternsOfSoftware.pdf`