О напряженности между объектно-ориентированным и обобщенным программированием в C++ (выдержка)

Томас Бекер, C++ Source, 15 октября 2007.

Теперь мы знаем, что идиомы обобщенного программирования и принципы объектно-ориентированного (ОО) дизайна имеют тенденцию конфликтовать. Я не верю, что прямой отказ от того или другого это очень практичное решение. Для того, чтобы успешно проектировать и реализовывать большие программные системы, мы должны использовать как ОО принципы, так и техники обобщенного программирования. То, что мы — и под «мы» я подразумеваю всех нас, C++ гуру, авторов, докладчиков, консультантов, так же как и программных инженеров в крупных компаниях — должны понимать, так это:

> Хорошая разработка включает в себя компромиссы на каждом направлении. Хороший, работающий, завершенный продукт никогда не является чистым с точки зрения какой-либо единственной идиомы или методологии. Искусство хорошей инженерии это не искусство изучения и применения одной верной идиомы без учета всех остальных. Искусство хорошей инженерии это знать возможные варианты, и затем разумно выбирать уступки вместо того, чтобы позволять другим делать это за себя.

Выходя за пределы этих размышлений, что этот учет компромиссов означает для примеров из предыдущего раздела? Что ж, компромиссы никогда не бывают простыми. Чаще всего это весьма неприятная вещь. Вот почему люди так любят эту чистую, безупречную, святая-святых абсолютную правду, которая проповедуется на каждом углу (и на многих конференциях по программному обеспечению также, к слову сказать).

Это правда, что когда бы вы ни раскрывали пару STL-итераторов, как например `std::vector<double>::const_iterator`, в интерфейсе своего класса, вы создаете зависимость времени компиляции. Более того, как мы видели в предыдущем разделе, вы необоснованно близко приближаетесь к нарушению основных принципов ОО программирования, таких как инкапсуляция и последовательный дизайн интерфейсов. Однако, вы также получаете и преимущество: когда компилятор делает надлежащую работу со встраиванием, ваши итераторы вектора скорее всего покажут такую же производительность, как обычные указатели. Этот выигрыш в производительности может оправдывать жертвование чистотой ОО. Важно понимать, что здесь пришлось пойти на уступку. Не такая уж редкость в научном программировании, когда выигрыш в производительности, полученный техниками обобщенного программирования, обязателен. Но также обыкновенно в больших программных системах, когда другие узкие места в производительности делают такой выигрыш неразличимым.

Мне кажется, что с тех пор как уступка между чистотой ОО-подхода и быстродействием была принята во внимание, маятник C++-сообщества сильно качнулся в сторону быстродействия. В 1990-х все было сосредоточено вокруг объектно-ориентированного анализа и дизайна. Любое упоминание вопросов производительности в отношении полиморфизма времени выполнения отклонялось как «старый C-стиль мышления». Сегодня все наоборот: любое предложение заменить или улучшить решение обобщенного программирования чем-то, что влечет за собой дополнительный вызов виртуальной функции, неизбежно встретит протест на основании того, что это «слишком медленно». Эта предсказуемая реакция не слишком полезна в реальной программной инженерии. Я всегда думал о том, что старая поговорка, касающаяся оптимизации и ее связи со злом, была достаточным руководством в этом смысле. Но, возможно, требуется уточнение:

> Оптимизация, эффект от которой ни один пользователь никогда не заметит, является корнем многих зол, в числе которых провалы программных проектов и следующие за ними провалы в бизнесе.

Теперь представьте, что вы попали в ту же ситуацию, что и я со своим плохим интерфейсом для «цифродробилки». Ваши клиенты требуют более ОО-совместимого дизайна, и они совершенно готовы заплатить штраф производительности в размере вызова виртуальной функции для каждой операции итератора. Это самое время подумать о «затирании типов» как о решении. Так что же такое «затирание типов», и как оно может помочь нам в этом случае?