Лекция № 6. О связи визуальных спецификаций с программных кодом.

В лекции объяснятся трудности автоматической генерации программного кода по визуальным моделям, обсуждаются отдельные области, где это возможно и стало общеупотребительной практикой – моделирование схем реляционных баз данных, структуры объектно-ориентированных приложений, структуры и поведения систем реального времен, а также бизнес-процессов компаний. Вводятся понятия возвратного проектирования (reverse engineering), циклической разработки (round-trip engineering) и управления согласованностью (inconsistency management).

Мечта об автоматической генерации кода по моделям. Известно, что как только программное обеспечение стало проникать в уже устоявшиеся промышленные области (производство автомобилей, самолетов, военных систем, бытовой техники и т.д.), то есть целевые системы стали содержать программные модули, которые сильно повысили возможности таких систем, то непредсказуемость разработок, а также качество конечных систем стали серьезной проблемой. Существенно повысился также процент неудавшихся проектов.

Один из способов преодолеть эту, надо сказать, глобальную проблему – это создавать системы в терминах той предметной области, для которой ПО предназначается, либо входит как составная часть программно-аппаратных комплексов. Это позволило бы создавать ПО прямо специалистам предметной области, с минимальным привлечением программистов, то есть без перерастаний одной из частей процесса разработки системы в отдельный и плохо предсказуемый процесс. Целевой программный код хочется получать в этой ситуации автоматически, по высокоуровневым спецификациям, а сами спецификации выполнять с помощью визуального моделирования. Ведь визуальные модели более наглядны, более понятны людям, в том числе не программистам – так почему бы их не использовать как средства высокоуровнего конструирования систем?

Например, если бизнес-компании нужно создать новую информационную систему, то ее специалисты-аналитики подробно описывают функции такой системы, и после этого нужное ПО создается автоматически. Или если необходим новая встроенная программно-аппаратная система для новой марки автомобиля, инженеры описывают его функции и архитектуру, а дальше все необходимое создается автоматически.

Об эволюции средств программирования. Идея **об** автоматической генерации программного кода по более высокоуровневым спецификациям – визуальным моделям – хорошо соответствует эволюции средств программирования. Схема этой эволюции представлена на рис. 6.1.

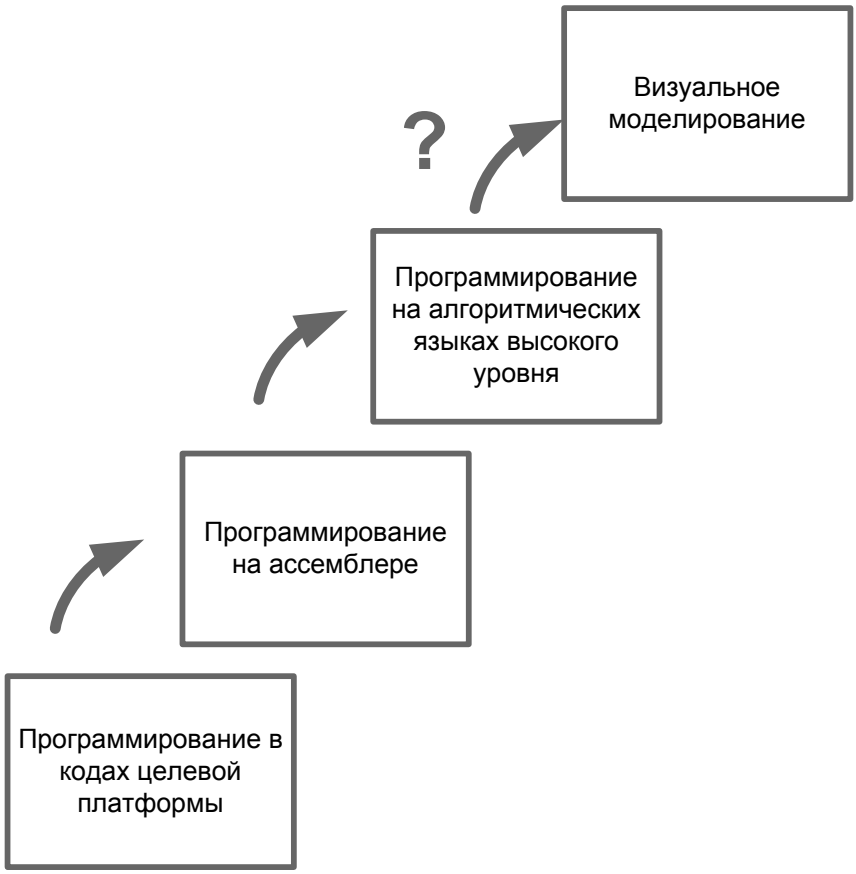

Рис. 6.1.

Программирование началось с создания исполняемых ЭВМ кодов «вручную». То есть то, что машина обрабатывала с помощью электронных схем, подавалось ей на вход с использованием простейших средств ввода, например, картонных перфокарт. Обработка введенных в машину программ, предшествующая их непосредственному исполнению, была минимальна.

Далее появились ассемблер-языки, позволяющие описывать программу не с помощью нулей и единиц, а посредством мнемонических команд, имеющих буквенные имена, численные параметры и т.д., а к целевые коды ассемблер-программы переводились компиляторами. Это существенно облегчило процесс программирования, но все равно программист должен был очень хорошо представлять внутреннее устройство целевой ЭВМ: количество регистров и их имена, размеры оперативной памяти и то, как именно данная программа будет ее использовать, все периферийные устройства и правила работы с ними и т.д.

Далее появились алгоритмические языки программирования – COBOL, Fortran, PL/1, Algol60, C, Pascal, C++, Java, #C и многие другие. По мере их развития программист получал возможность все меньше и меньше задумываться о деталях процесса выполнения программы на ЭВМ и все больше внимания уделять описанию логики задачи, которую реализовывала его программа. Соответственно, спектр задач, которые становилось способным решать ПО, существенно расширялся, сложность программ увеличивалась.

В этой цепочке каждый следующий шаг, практически, вытеснял предыдущий: в кодах целевых ЭВМ перестали массового программировать, когда появились ассемблер-языки и их реализации для разных платформ. Целевые коды стали генерироваться по ассемблер-спецификациям автоматически. Далее, алгоритмические языки высокого уровня столь же радикально вытеснили программирование на ассемблере, и целевой код автоматически генерировался теперь уже по текстам на этих языках. При желании генерируется и текст на ассемблере, если есть необходимость проанализировать результаты генерации (ведь тексты на ассемблере, с одной стороны, существенно легче воспринимаются, чем целевой код, с другой стороны, очень они близки к нему).

Итак, можно сказать, что эволюция средств программирования двигалась от вычислителя к человеку с сохранением связи с вычислителем. Ведь, с одной стороны, программы ориентированы на вычислитель, который их должен исполнять. Поэтому они должны содержать максимально точную и полную информацию о том, как именно вычислительно должен их выполнять. С другой стороны, программы должны быть удобны для разработки человеком. Ведь держать в уме причудливо ветвящийся, пересекающий, сходящийся и расходящийся вновь поток управления с сотнями операций, переменных и т.д. человеку не возможно, очевидно, что после достижения определенного уровня наступает предел. А если в программировании участвует несколько человек, то им для общения необходимы абстракции, более удобные, чем машинные команды. Наконец, программы часто передаются на сопровождение другим людям, в коллектив разработчиков входят новые сотрудники, наконец, сами программисты после перерыва, обращаясь к своим программам, с трудом в них разбираются. Таким образом, программы должны быть максимально доступны для человека – как для самого автора, чтобы упростить процесс его работы, так и для других людей, чтобы сделать работу одного более понятной другим. Эта двойственная природа программ отражена на рис. 6.2.

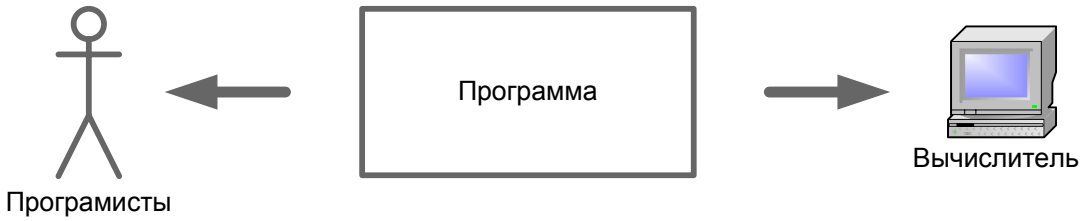

Рис. 6.2.

Визуальные моделей: понятность или исполняемая семантика? Ожидалось, что визуальное моделирование станет следующим этапом, заменив собой алгоритмические языки и предоставив более широкие возможности программистам. Однако оказалось, что визуальные модели, действительно удобные в работе над ПО, склонны терять исполняемую семантику. Другими словами, информация, которая в них содержится, недостаточно полна и детальна, чтобы по ней вычислитель мог бы выполнить вычисление. Ведь никакая «умная» генерация не может добавить того, что нет изначально. Если же визуальные модели усложнять с тем, чтобы они были пригодны для использования вычислителем, то в общем случае они теряют наглядность и становятся бесполезными. Кому из разработчиков нужны непонятные, но правильные описания программного обеспечения? Есть тексты на языках программирования, есть документы, есть возможность спросить и узнать….

Таким образом, существует семантический разрыв между визуальными моделями и программами, как показано на рис. 6.3. И он препятствует автоматической генерации

программного кода и тому, чтобы визуальное моделирование прочно стало следующим шагом в развитии средств программирования, как показано на рис. 6.2.

То, что визуальное моделирование не стало «серебряной пулей» при разработке ПО, объясняется сложностью и невидимостью ПО, как это обсуждалось в предыдущих лекциях. Не существует естественного способа визуального представления невидимого, иначе, чем с помощью специальных текстов, то есть программ, созданных на языках программирования. Более того, в общем случае невозможно автоматически генерировать ПО по каким бы то ни было спецификациям в терминах предметной области.

Визуальное моделирование –не единственный подход, с помощью которого делались попытки повысить уровень абстракции средств программирования. Можно указать, например, на Intentional Programming и Aspect-Oriented programming [4] .

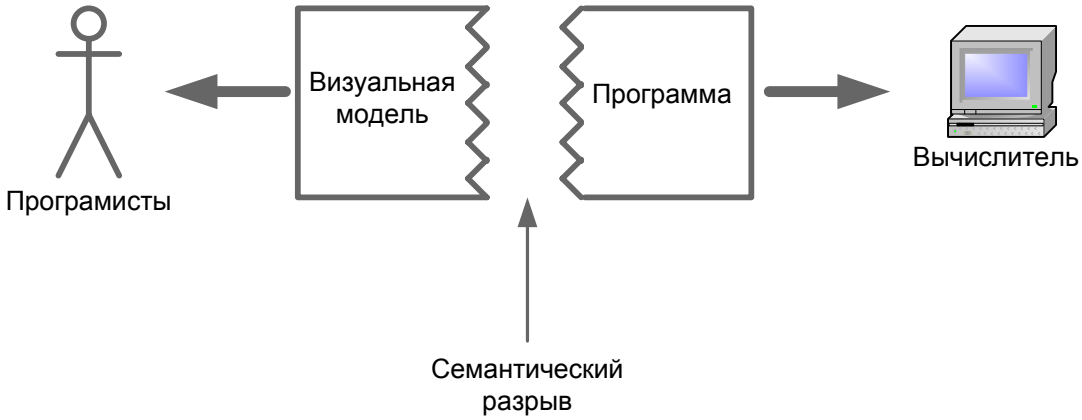

Рис. 6.3.

Где выход? Неудача решения вопроса с генерацией «в лоб», универсально, не говорит о том, что это не возможно в принципе. Необходимо лишь понизить универсальность запроса:

1. *Генерационные решения в рамках product lines.* Создавать специальные кодогенераторы по визуальным моделям для отдельного проекта, максимально учитывая его специфику, то есть делая тот самый «умный» генератор, который не возможен в общем случае. Но поскольку создание таких генераторов для каждого проекта – дорогостоящее дело (если проект не супергигантский), то подобные специализированные средства лучше повторно использовать в рамках разработки похожих систем. Этот вывод хорошо согласуется с известным подходом к разработке ПО методом организации семейств программных продуктов (product line approach).

2. *Ограниченная генерация.* Автоматически генерировать отдельные виды ПО, компоненты и др. фрагменты. Среди таких видов мы выделим:
   - схемы реляционных баз данных;
   - структура объектно-ориентированных приложений;
   - структура и поведение систем реального времени;
   - описание бизнес-процессов компаний.

3. *Генерация прототипов.* Возможно также использовать автоматическую генерацию по моделям для создания прототипа системы. Сгенерированный по предварительным описаниям прототип системы в дальнейшем дорабатывается «в ручную».

4. *Отказ от автоматической генерации.* генерации Наконец, от автоматической генерации кода по визуальным моделям можно отказаться вовсе, используя модели лишь как документы – на ранних стадиях разработки, при общении, в презентациях, в документации к ПО (в основном, при описании ПО).

Общая структура генерационных решений.

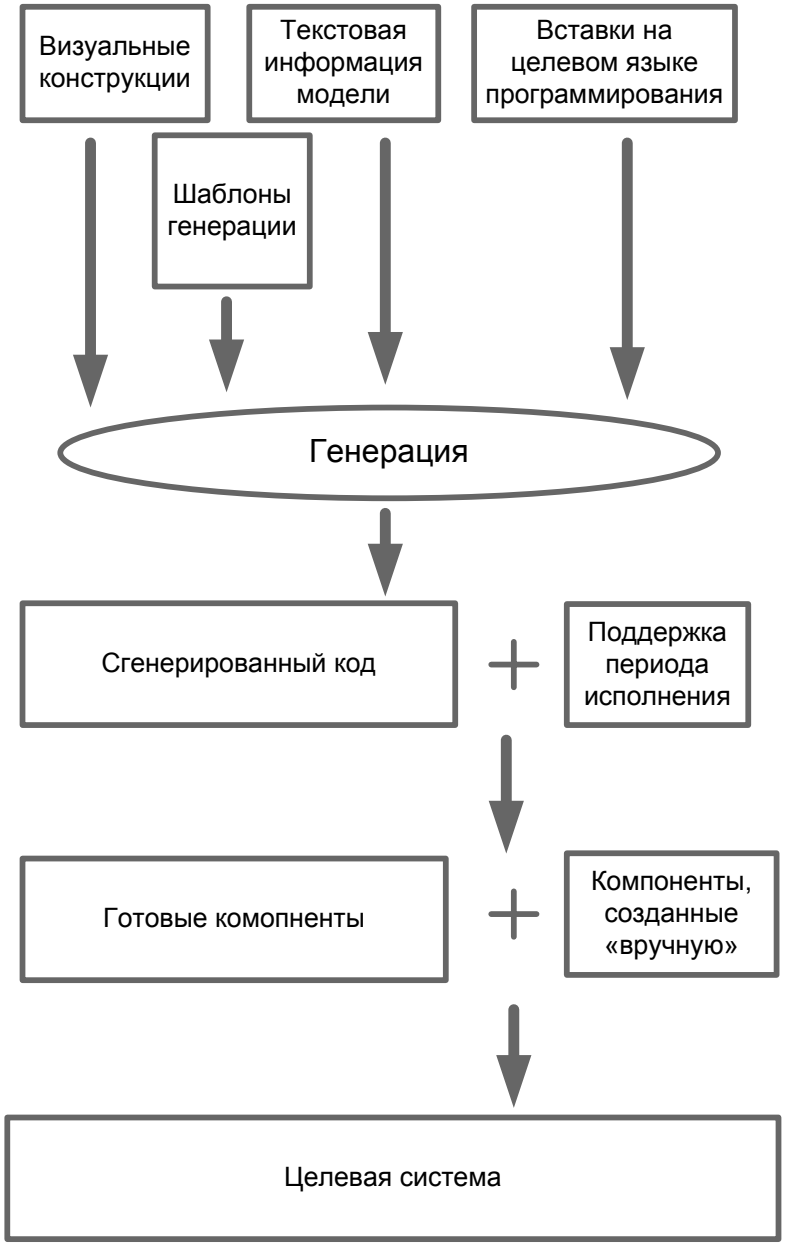

Рис. 6.4.

Другие способы связи визуальных моделей и программного кода. Необходимо заметить, что коль скоро автоматическая генерация программного кода по визуальным моделям не стала общепринятым способом разработки ПО, то вопрос связи диаграмм с кодом стал более сложным.

Во-первых, возникала задача возвратного проектирования (reverse engineering) – частью процесса сопровождения ПО и позволяющая понять систему с тем, чтобы внести в нее соответствующие изменения. То есть речь идет о восстановлении верхеуровневой информации о системе, например, требований, архитектуры и т.д. При этом предпочтителен графический вид этой верхнеуровнвой информации.

Во-вторых, возникает задача циклической разработки (roundtrip engineering) – возможность средств разработки поддерживать целостность различных, изменяемых артефактов разработки ПО во время процесса его создания и эволюции. То есть эти часть этих артефактов может изменяться естественным путем, при этом необходимые изменения будут распространены во все другие, связанные артефакты. Как правило, в рамках циклической разработки рассматривают соответствие визуальных моделей и программного кода. Задача поддержки целостности различных UML-моделей в рамках одного проекта носит название управление согласованностью (inconsistency management).

Основная литература

1. Д.В. Кознов. Языки визуального моделирования: проектирование и визуализация программного обеспечения. Учебное пособие. – Изд-во СПбГУ, 2004. – 143 с.
2. Зверева В.А., Кознов Д.В., Бережной А.С. Обзор подходов к управлению согласованностью активов разработки ПО при использовании UML // Готовится к печати в «Системное программирование». Вып.2: Сб. статей. – СПб.: Изд-во СПБГУ, 2006.
3. F. Brooks. No Silver Bullet. – Information Proceeding of the IFIP 10th World Computing Conference. 1986. – P. 1069-1076. (Русский перевод: Ф. Брукс. Мифический человеко-месяц или как создаются программные системы. – СПб Символ, 2000).

Дополнительная литература

4. K.Czarnecki, U.W.Eisenecker. Generative programming: Methods, Tools, and Applications. – Addison-Wesley 2000. – 832 p.
5. E.1. Chikofsky, J. H. Cross. Reverse Engineering and Design Recovery: Taxonomy. – IEEE Software, January 1990. – P. 13 -17.
6. S. Sendall, J. Kuster. Taming Model Round-Trip Engineering. – Proceedings of Workshop on Best Practices for Model-Driven software Development (part of 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, Applications), Vancouver, Canada, 25th October 2004.
7. M. Elaasar, L. Briand. An Overview of UML Consistency Management. – Technical Report SCE-04-18. Department of Systems and Computer Engineering. – 2004.