УДК 004.05

СПОСОБЫ ФОРМАЛЬНОЙ СПЕЦИФИКАЦИИ ПРИНЦИПОВ ПРОЕКТИРОВАНИЯ ПРОГРАММНЫХ СРЕДСТВ

В. В. Бураков,
канд. техн. наук, доцент
Санкт-Петербургский государственный университет аэрокосмического приборостроения

Представлены два способа формализации принципов проектирования, основанные на вычислении значений метрик и формировании множества ролей элементов программ, с помощью которых можно реализовать более точную оценку качества программных средств и выработать более обоснованные действия для его улучшения.

Введение

Скорость развития технологий разработки программных средств (ПС) бортовых систем в целом не отвечает скорости роста функциональных требований и сложности программ. При использовании существующих возможностей для внесения изменений в ПС бортовой системы требуются месяцы, порой годы интенсивной дорогостоящей работы [1]. Сложность процесса сопровождения бортовых ПС во многом обуславливается особыми требованиями, предъявляемыми к их качеству. Для управления качеством ПС наиболее важной информацией является причинно-следственная, адекватное представление которой возможно только за счет учета принципов проектирования. Существующие модели качества не регламентируют четкое соотнесение показателей качества и практики проектирования в рамках какой-либо парадигмы, иными словами, отсутствует методика формализации принципов проектирования. Отсюда происходит разрыв между тем, что измеряется, как интерпретируется и какие преобразования ПС для улучшения его качества можно было бы применить [2].

Принципы проектирования являются контекстно-зависимыми понятиями, отражающими основные подходы к решению задач на определенных парадигме и языке программирования. Принципы проектирования основываются на опыте, накопленном при разработке успешных проектов разных масштабов с использованием определенной парадигмы программирования (например, процедурной, функциональной, объектно-ориентированной). Значение, которое оказывает на качество ПС систематизированное применение определенных принципов проектирования, стало предметом исследований. Многие авторы пытались синтезировать положительный опыт проектирования в шаблонах [3], другие концентрировали внимание на отрицательных — рефакторинге [4], антипаттернах [5] и дефектах. Авторы этих исследований формулируют принципы проектирования на разных уровнях — от точного, например, «класс не должен агрегировать более шести объектов», до неопределенного «необходимо избегать централизации управления». Подобная разобщенность затрудняет применение этих принципов согласованным и систематизированным образом. Зачастую те принципы, которые удается сформулировать только на абстрактном уровне, оказывают большее влияние на качество, чем точно формулируемые. Предлагаемая статья посвящена раскрытию формального подхода к описанию принципов проектирования.

Модель ПС

Для моделирования программных сущностей и формализации метрик используются ориентированные помеченные типизированные графы.

Ориентированный граф $G = (V, E, s, t)$ состоит из двух множеств: конечного множества V, элементы которого называются вершинами (ребрами), и конечного множества E, элементы которого называются ребрами (дугами).

Помеченный граф. Пусть $L = (VL, EL)$, $A = (VA)$ — пара непересекающихся потенциально бесконечных множеств меток и ролей соответственно. (L, A)-помеченный граф G представляет собой тройку (g, l, a) такую, что имеет место: $g = (V, E, s, t)$ — граф; $l = (vl: V \to VL, el: E \to EL)$ — пара функций пометки соответственно вершин и ребер, при этом vl является инъективной; $a = (va: V \to VA)$ — функция отображения вершин на множество ролей; определяется также инъективная функция edge: $e \to (el(e), s(e), t(e))$.

Метки служат для идентификации вершин и ребер. Роли описывают контекст использования сущности, при этом каждая из сущностей играет

в рамках структуры связей определенную роль. На сущность не накладывается ограничений по уникальности выполняемой роли. С точки зрения разных проектных решений, одна и та же сущность может быть задействована в разных ролях.

Помеченный типизированный граф. Пусть $T = (VT, ET)$ — пара непересекающихся конечных множеств предопределенных типов вершин и ребер. (L, A)-помеченный T-типизированный граф G является двойкой (g, type) такой, что g — (L, A)-помеченный граф и $\text{type} = (vt: V \to VT, et: E \to ET)$ — пара функций, связывающих соответственно с каждой вершиной и ребром его тип. Для графа G и вершины $v \in V_G$ существуют:

— множество входных ребер $IE_G(v) = \{e \in E_G: t(e) = v\}$;

— множество выходных ребер $OE_G(v) = \{e \in E_G: s(e) = v\}$;

— множество входных вершин $IV_G(v) = \{t(e): e \in IE_G(v)\}$;

— множество выходных вершин $OV_G(v) = \{s(e): e \in OE_G(v)\}$.

Моделирование Java-кода

Для формального представления кода или модели конкретного проекта с помощью графов необходимо выполнить следующие шаги по адаптации: идентифицировать типы вершин и ребер, определить частичный порядок вершин и ребер, описать аксиомы для вершин и ребер [2]. Следует подчеркнуть, что при решении задачи формализации принципов проектирования нет необходимости в моделировании всех синтаксических аспектов языка моделирования или кодирования. Необходимое и достаточное множество языковых конструкций, подлежащих моделированию, определяется целью создания моделей — в описываемом случае в качестве такой цели выступает формализация принципов проектирования. Множество $V_G = \{A, C, F, O, K, L\}$ всех возможных типов вершин служит для представления соответственно абстрактного класса, класса, открытого поля, метода, открытого метода доступа или модификации значения поля, оператора цикла. Множество $E_G = \{i, u, o, c, a\}$ всех возможных типов ребер представлено в таблице.

Тип	Описание
j: $C \to A$	Наследование класса от абстрактного класса
j: $C \to C$	Наследование одного класса от другого
i: $C \to C$	Использование абстрактного предка класса
u: $C \to C$	Использование класса
o: $F \to C$	Принадлежность поля классу
o: $O \to C$	Принадлежность метода классу
o: $K \to C$	Принадлежность открытого метода доступа или модификации значения поля классу
c: $C \to C$	Классовый тип поля
a: $O \to F$	Доступ к полю из метода
a: $K \to C$	Использование поля открытым методом доступа или модификации
o: $L \to C$	Принадлежность оператора цикла методу
o: $L \to L$	Вложенность операторов циклов

Квантификация принципов проектирования

Под квантификацией принципов проектирования понимается следующая совокупность процессов:

1) формирование базовых метрических пространств — выбор множества базовых и производных метрик, влияющих на принцип проектирования;

2) определение эталонных значений — выбор интервалов значений множества базовых и производных метрик, при соблюдении которых не происходит нарушения принципа проектирования;

3) построение метрического пространства принципа проектирования — построение производного метрического пространства на основе аналитического выражения над множеством метрик;

4) формирование эталонного и дефектного подпространств — определение эталонных и дефектных интервалов значений производной метрики, выражающей принцип проектирования.

Эти четыре шага являются в основном результатом экспертной работы. Учитываются хорошо зарекомендовавшие практики, накопленный опыт разработчиков. Квантифицированные таким образом принципы проектирования затем нуждаются в уточнении. Для этого создаются репозитории проектов, позволяющие на основе статистики и применения методов оптимизации уточнить выбор метрик, добиться более правильных значений формул и скорректировать эталонные интервалы. С помощью квантифицированных принципов проектирования производится фильтрация кода (модели), целью которой может быть не только поиск дефектов, но и решение других задач, например поиск фрагментов кода, подлежащих повторному использованию, поиск шаблонов проектирования в коде, поиск фрагментов кода для рефакторинга.

В целях поиска фрагментов кода, обладающих определенными свойствами, для базовых и производных метрик устанавливаются фильтры критических значений. Для каждого фильтра посредством комплексирования базовых и производных показателей определяется новая производная метрика с фиксированным набором допустимых или недопустимых значений.

Следует отметить, что качество квантификации принципов проектирования зависит как от адекватного набора метрик, так и от правильно выбранного диапазона их значений. В качестве механизмов оптимизации диапазонов значений метрик целесообразно применять генетические алгоритмы и системы самообучения.

Приведем пример квантификации принципов проектирования. В качестве квантифицируемого принципа будем использовать принцип объектно-ориентированного проектирования «увеличение сцепления».

Сцепление — это степень использования методами класса методов и атрибутов того же класса. При высоком сцеплении обязанности класса тесно связаны между собой, класс в большей мере концептуально монолитен. Класс с низкой степенью сцепления выполняет разнородные действия и не связанные между собой обязанности, концептуально разобщен. «Увеличение сцепления» — это принцип проектирования, согласно которому каждый класс должен моделировать одно концептуальное понятие и стремиться к внутренней целостности. Выберем следующий набор метрик для квантификации этого принципа проектирования:

— количество пар методов, которые не используют общие поля класса:

$$\mu_1 = \sum_{\substack{m_1, m_2 \in \{s(e) \in IV_G(v):\\ vt(s(e))=O\} \wedge \\ |OV_G(m_1) \cap OV_G(m_2)|=1}} |\{m_2\}|;$$

— количество пар методов, которые используют общие поля класса:

$$\mu_2 = \sum_{\substack{f \in \{s(e) \in IV_G(v):\\ vt(s(e))=F\}}} \frac{ivec(f, O, a)!}{2!(ivec(f, O, a)-2)!},$$

где $ivec(v, t, r) = \left| \begin{matrix} \{t(e) : e \in IV_G(v) \wedge vt(s(e)) = t \\ \wedge et(e) = r\} \end{matrix} \right|$;

— количество методов класса, которые не используют поля класса:

$$\mu_3 = \left| \{m \in \{s(e) \in IV_G(v) : vt(s(e)) = O\} \wedge |OV_G(m)| = 1\} \right|;$$

— количество пар методов, которые не используют общие поля класса, минус количество пар методов, которые используют общие поля класса: $\mu_4 = \mu_1 - \mu_2$.

Для оценки того, насколько класс соответствует принципу «увеличение сцепления», введем следующую формулу над указанными метриками:

$$\mu_{hc} = \frac{3\mu_4 + \mu_3}{4}.$$

Затем определим эталонный интервал этой метрики — [0,100, ∞]. Если для некоторого класса выполняется $\mu_{hc} \in [0{,}100, \infty]$, то будем считать, что структура класса соответствует принципу проектирования «увеличение сцепления».

Описание принципов проектирования с помощью ролей

Как уже отмечалось, с помощью ролей вершин графов можно вводить правила проектирования, описывающие типовые ограничения на возможные зависимости программных сущностей. Этим способом можно специфицировать нежелательные с точки зрения качества ПС проектные решения.

Варианты описания ролей в программном коде

Для того чтобы использовать роли и правила использования ролей, необходимо аннотировать программные сущности идентификаторами ролей. В зависимости от парадигмы программирования, конкретного языка способы аннотации могут быть разными. Для объектно-ориентированной парадигмы варианты для сопоставления ролей с программными сущностями могут быть следующими.

Введение собственных тегов или атрибутов. Этот вариант предполагает использование зависящей от языка программирования техники комментирования для введения идентификаторов ролей программных сущностей. Например, для языка Java спецификация того, что некоторый класс выполняет роли объекта предметной области из архитектурного шаблона «модель—представление—контроллер» и роль субъекта из поведенческого шаблона «наблюдатель», могла бы выглядеть так:

```
/** @...
  @role domainObject
  @role subject
…
*/
public class Person {…}
```

Спецификация тех же ролей для класса на C# использует технику документирования, определенную для этого языка:

```
/// <roles>
/// domainObject,
/// subject
/// </roles>
public class Person {…}
```

Использование правил именования. Другим способом идентификации ролей является применение особых правил именования программных сущностей:

```
public class PersonDO {…}
```

Этот способ является менее удачным, так как ухудшает читабельность кода, особенно если сущность выполняет несколько ролей, смешивая обязанности сущности с информацией о способе реализации этих обязанностей.

Распространение ролей агрегата на составляющие его элементы. Для модульных языковых структур, например пакетов (Java) или пространств имен (C#), целесообразно использовать собственные теги комментариев для идентификации ролей и распространять информацию о ролях на все элементы этого модуля. Например, для идентификации ролей архитектурных шаблонов в силу соответствия распределения по модулям выполняемым ролям.

Проиллюстрируем механизм использования ролей и правил на примере шаблонов (типовых решений) в области объектно-ориентированного дизайна [3]. Выберем шаблон «наблюдатель» и условимся, что при генерации графа ПС, в том случае, если методы одного объекта программы ис-

пользуют доступ к полям или методам другого объекта программы, между вершинами, представляющими эти объекты, порождаются ребра, имеющие тип «использование», направленные от первого объекта ко второму.

«Наблюдатель» — шаблон, позволяющий создавать объекты, зависимые от данного, которые получают извещения при каждой смене состояния первичного объекта [3]. Шаблон определяет зависимость типа один ко многим между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом и автоматически обновляются.

Диаграмма классов, отражающая структуру шаблона «наблюдатель», представлена на рис. 1.

Классы шаблона:

— Subject — субъект, располагает информацией о своих наблюдателях. За субъектом может следить любое число наблюдателей; предоставляет интерфейс для присоединения и отделения наблюдателей;

— Observer — наблюдатель, определяет интерфейс обновления для объектов, которые должны быть уведомлены об изменении субъекта;

— ConcreteSubject — конкретный субъект, сохраняет состояние, представляющее интерес для конкретного наблюдателя ConcreteObserver; посылает информацию своим наблюдателям, когда происходит изменение;

— ConcreteObserver — конкретный наблюдатель, хранит ссылку на объект класса ConcreteSubject; сохраняет данные, которые должны быть согласованы с данными субъекта; реализует интерфейс обновления, определенный в классе Observer, чтобы поддерживать согласованность с субъектом.

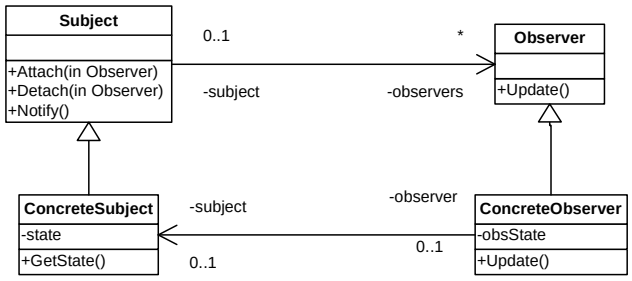
Отношения классов:

— объект ConcreteSubject уведомляет своих наблюдателей о любом изменении, которое могло бы привести к рассогласованности состояний наблюдателя и субъекта;

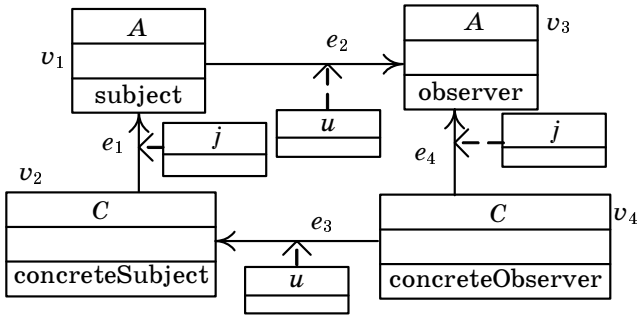
— после получения от конкретного субъекта уведомления об изменении объект ConcreteObserver может запросить у субъекта дополнительную информацию, которую использует для того, чтобы оказаться в состоянии, согласованном с состоянием субъекта.

Графовая модель шаблона «наблюдатель» показана на рис. 2.

Главное достоинство шаблона «наблюдатель» состоит в ослаблении связей компонент системы. Для правильного использования этого шаблона необходимо, чтобы субъект имел информацию лишь о том, что у него есть ряд наблюдателей, каждый из которых подчиняется простому интерфейсу абстрактного класса Observer. Субъекту не должны быть известны конкретные классы наблюдателей. Для формализации этого правила введем роли класса субъекта — subject и классов конкретных наблюдателей — concreteObserver. Сформулируем правило проектирования кода с использованием шаблона «наблюдатель» формально: для любого v: $va(v)$ = subject и любого w: $va(w)$ = concreteObserver должно выполняться $(v, w) \notin (IV_G)^n$.



■ ***Рис. 1.*** *Структура шаблона «наблюдатель»*



■ ***Рис. 2.*** *Графовая модель шаблона «наблюдатель»*

Заключение

Принципы проектирования устанавливают для каждой парадигмы границы качества ПС. Использование предлагаемого формального описания принципов проектирования позволит учесть в моделях качества не только статистическую, но и причинно-следственную информацию, сделает оценку качества ПС более точной, явится основой для формирования более действенного набора мероприятий по улучшению качества ПС.

Литература

1. **Cooke D. E., Barry M., Lowry M., Green C.** NASA's Exploration Agenda and Capability Engineering // Computer. 2006. Vol. 39. N 1. P. 63–73.
2. **Бураков В. В.** Формализация процесса преобразований программного обеспечения // Управление и информатика в авиакосмических системах. Приложение к журналу «Мехатроника, автоматизация, управление». 2006. № 11. С. 19–24.
3. **Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.** Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2007. 366 с.
4. **Фаулер М.** Рефакторинг: улучшение существующего кода. СПб.: Символ-Плюс, 2004. 432 с.
5. **Тейт Б.** Горький вкус Java. СПб.: Питер, 2003. 333 с.